Computer Graphics

PRINCIPLES AND PRACTICE

Foley • van Dam • Feiner • Hughes

SECOND EDITION in C

THE SYSTEMS PROGRAMMING SERIES

Computer Graphics

PRINCIPLES AND PRACTICE

SECOND EDITION in C

Foley • van Dam • Feiner • Hughes

James D. Foley (Ph.D., University of Michigan) is the founding director of the interdisciplinary Graphics, Visualization & Usability Center at Georgia Institute of Technology, and Professor of Computer Science and of Electrical Engineering. Coauthor with Andries van Dam of *Fundamentals of Interactive Computer Graphics*. Foley is a member of ACM, ACM SIGGRAPH, ACM SIGCHI, the Human Factors Society, IEEE, and the IEEE Computer Society. He recently served as Editor-in-Chief of *ACM Transactions on Graphics*, and is on the editorial boards of *Computers and Graphics*, *User Modeling and User-Adapted Interaction*, and *Presence*. His research interests include model-based user interface development tools, user interface software, information visualization, multimedia, and human factors of the user interface. Foley is a Fellow of the IEEE, and a member of Phi Beta Kappa, Tau Beta Phi, Eta Kappa Nu, and Sigma Xi. At Georgia Tech, he has received College of Computing graduate student awards as Most Likely to Make Students Want to Grow Up to Be Professors, Most Inspirational Faculty Member, the campus Interdisciplinary Activities Award, and the Sigma Xi Sustained Research Award.

Andries van Dam (Ph.D., University of Pennsylvania) was the first chairman of the Computer Science Department at Brown University. Currently Thomas J. Watson, Jr. University Professor of Technology and Education and Professor of Computer Science at Brown, he is also Director of the NSF/ARPA Science and Technology Center for Computer Graphics and Scientific Visualization. His research interests include computer graphics, hypermedia systems, and workstations. He is past Chairman of the Computing Research Association, Chief Scientist at Electronic Book Technologies, Chairman of Object Power's Technical Advisory Board, and a member of Microsoft's Technical Advisory Board. A Fellow of both the IEEE Computer Society and of ACM, he is also cofounder of ACM SIGGRAPH. Coauthor of the widely used book *Fundamentals of Interactive Computer Graphics* with James Foley, and of *Object-Oriented Programming in Pascal: A Graphical Approach*, with D. Brookshire Conner and David Niguidula, he has, in addition, published over eighty papers. In 1990 van Dam received the NCGA Academic Award, in 1991, the SIGGRAPH Steven A. Coons Award, and in 1993 the ACM Karl V. Karlstrom Outstanding Educator Award.

continued on the back flap

continued from the front flap

Steven K. Feiner (Ph.D., Brown University) is Associate Professor of Computer Science at Columbia University, where he directs the Computer Graphics and User Interfaces Lab. His current research focuses on 3D user interfaces, virtual worlds, augmented reality, knowledge-based design of graphics and multimedia, animation, visualization, and hypermedia. Dr. Feiner is on the editorial boards of *ACM Transactions on Graphics*, *IEEE Transactions on Visualization and Computer Graphics*, and *Electronic Publishing*, and is on the executive board of the IEEE Technical Committee on Computer Graphics. He is a member of ACM SIGGRAPH and the IEEE Computer Society. In 1991 he received an ONR Young Investigator Award. Dr. Feiner's work has been published in over fifty papers and presented in numerous talks, tutorials, and panels.

John F. Hughes (Ph.D., University of California, Berkeley) is an Assistant Professor of Computer Science at Brown University, where he codirects the computer graphics group with Andries van Dam. His research interests are in applications of mathematics to computer graphics, scientific visualization, mathematical shape description, mathematical fundamentals of computer graphics, and low-dimensional topology and geometry. He is a member of the AMS, IEEE, and ACM SIGGRAPH. His recent papers have appeared in *Computer Graphics*, and in Visualization Conference Proceedings. He also has a long-standing interest in the use of computer graphics in mathematics education.

RELATED TITLES

The Pascal-language version of *Computer Graphics: Principles and Practice, Second Edition* remains available. ISBN 0-201-12110-7

Introduction to Computer Graphics is an abbreviated version of Foley, van Dam, Feiner, and Hughes that focuses on topics essential for beginners in the field and provides expanded explanations for readers with less technical background, also with examples in C. Added to the original list of authors, Richard Phillips was principally responsible for this adaptation. ISBN 0-201-60921-5

For information about other Addison-Wesley books, you may access our electronic catalog through the World Wide Web at http://www.aw.com/ or through gopher aw.com

ATI Ex. 2003
IPR2023-00564
Page 2 of 241

THE
SYSTEMS
PROGRAMMING
SERIES

Computer Graphics: Principles and Practice

SECOND EDITION in C

Foley • van Dam • Feiner • Hughes

ATI Ex. 2003
IPR2023-00564
Page 3 of 241

SECOND EDITION IN C

Computer Graphics

PRINCIPLES AND PRACTICE

ATI Ex. 2003
IPR2023-00564
Page 4 of 241

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

ADDISON-WESLEY PUBLISHING COMPANY
Reading, Massachusetts • Menlo Park, California • New York
Don Mills, Ontario • Wokingham, England • Amsterdam • Bonn
Sydney • Singapore • Tokyo • Madrid • San Juan • Milan • Paris

ATI Ex. 2003
IPR2023-00564
Page 5 of 241

Sponsoring Editor: Peter S. Gordon
Production Supervisor: Bette J. Aaronson
Production Supervisor for the C edition: Juliet Silveri
Copy Editor: Lyn Dupré
Text Designer: Herb Caswell
Technical Art Consultant: Joseph K. Vetere
Illustrators: C&C Associates
Cover Designer: Marshall Henrichs
Manufacturing Manager: Roy Logan

This book is in the **Addison-Wesley Systems Programming Series**
Consulting editors: IBM Editorial Board

Library of Congress Cataloging-in-Publication Data

Computer graphics: principles and practice / James D. Foley . . . [et al.]. — 2nd ed. in C.
p. cm.
Includes bibliographical references and index.
ISBN 0-201-84840-6
1. Computer graphics. I. Foley, James D., 1942–
T385.C5735 1996
006.6'6—dc20 95-13631
CIP

Reprinted with corrections, July 1997.

Cover: "Dutch Interior," after Vermeer, by J. Wallace, M. Cohen, and D. Greenberg, Cornell University (Copyright © 1987 Cornell University, Program of Computer Graphics.)

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and Addison-Wesley was aware of a trademark claim, the designations have been printed in initial caps or all caps.

The programs and applications presented in this book have been included for their instructional value. They are not guaranteed for any particular purpose. The publisher and the author do not offer any warranties or representations, nor do they accept any liabilities with respect to the programs or applications.

Reprinted with corrections November 1992, November 1993, and July 1995.

Copyright © 1996, 1990 by Addison-Wesley Publishing Company, Inc.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher. Printed in the United States of America.

6 7 8 9 10-DOC-99 98 97

Contents

CHAPTER 1
INTRODUCTION 1

1.1 Image Processing as Picture Analysis 2
1.2 The Advantages of Interactive Graphics 3
1.3 Representative Uses of Computer Graphics 4
1.4 Classification of Applications 6
1.5 Development of Hardware and Software for Computer Graphics 8
1.6 Conceptual Framework for Interactive Graphics 17
1.7 Summary 21
Exercises 22

CHAPTER 2
PROGRAMMING IN THE SIMPLE RASTER GRAPHICS PACKAGE (SRGP) 25

2.1 Drawing with SRGP 26
2.2 Basic Interaction Handling 40
2.3 Raster Graphics Features 52
2.4 Limitations of SRGP 60
2.5 Summary 63
Exercises 64

CHAPTER 3
BASIC RASTER GRAPHICS ALGORITHMS FOR DRAWING 2D PRIMITIVES 67

3.1 Overview 67
3.2 Scan Converting Lines 72
3.3 Scan Converting Circles 81
3.4 Scan Converting Ellipses 88
3.5 Filling Rectangles 91
3.6 Filling Polygons 92
3.7 Filling Ellipse Arcs 99
3.8 Pattern Filling 100
3.9 Thick Primitives 104
3.10 Line Style and Pen Style 109
3.11 Clipping in a Raster World 110
3.12 Clipping Lines 111

3.13 Clipping Circles and Ellipses 124
3.14 Clipping Polygons 124
3.15 Generating Characters 127
3.16 SRGP_copyPixel 132
3.17 Antialiasing 132
3.18 Summary 140
Exercises 142

CHAPTER 4
GRAPHICS HARDWARE **145**

4.1 Hardcopy Technologies 146
4.2 Display Technologies 155
4.3 Raster-Scan Display Systems 165
4.4 The Video Controller 179
4.5 Random-Scan Display Processor 184
4.6 Input Devices for Operator Interaction 188
4.7 Image Scanners 195
Exercises 197

CHAPTER 5
GEOMETRICAL TRANSFORMATIONS **201**

5.1 2D Transformations 201
5.2 Homogeneous Coordinates and Matrix Representation of 2D Transformations 204
5.3 Composition of 2D Transformations 208
5.4 The Window-to-Viewport Transformation 210
5.5 Efficiency 212
5.6 Matrix Representation of 3D Transformations 213
5.7 Composition of 3D Transformations 217
5.8 Transformations as a Change in Coordinate System 222
Exercises 226

CHAPTER 6
VIEWING IN 3D **229**

6.1 Projections 230
6.2 Specifying an Arbitrary 3D View 237
6.3 Examples of 3D Viewing 242
6.4 The Mathematics of Planar Geometric Projections 253
6.5 Implementing Planar Geometric Projections 258
6.6 Coordinate Systems 279
Exercises 281

CHAPTER 7
OBJECT HIERARCHY AND SIMPLE PHIGS (SPHIGS) 285

7.1 Geometric Modeling . . . 286
7.2 Characteristics of Retained-Mode Graphics Packages . . . 293
7.3 Defining and Displaying Structures . . . 295
7.4 Modeling Transformations . . . 304
7.5 Hierarchical Structure Networks . . . 308
7.6 Matrix Composition in Display Traversal . . . 315
7.7 Appearance-Attribute Handling in Hierarchy . . . 318
7.8 Screen Updating and Rendering Modes . . . 322
7.9 Structure Network Editing for Dynamic Effects . . . 324
7.10 Interaction . . . 328
7.11 Additional Output Features . . . 332
7.12 Implementation Issues . . . 334
7.13 Optimizing Display of Hierarchical Models . . . 340
7.14 Limitations of Hierarchical Modeling in PHIGS . . . 341
7.15 Alternative Forms of Hierarchical Modeling . . . 343
7.16 Summary . . . 345
Exercises . . . 346

CHAPTER 8
INPUT DEVICES, INTERACTION TECHNIQUES, AND INTERACTION TASKS 347

8.1 Interaction Hardware . . . 349
8.2 Basic Interaction Tasks . . . 358
8.3 Composite Interaction Tasks . . . 381
Exercises . . . 388

CHAPTER 9
DIALOGUE DESIGN 391

9.1 The Form and Content of User-Computer Dialogues . . . 392
9.2 User-Interface Styles . . . 395
9.3 Important Design Considerations . . . 403
9.4 Modes and Syntax . . . 414
9.5 Visual Design . . . 418
9.6 The Design Methodology . . . 429
Exercises . . . 431

CHAPTER 10
USER INTERFACE SOFTWARE 435

10.1 Basic Interaction-Handling Models . . . 436
10.2 Window-Management Systems . . . 439
10.3 Output Handling in Window Systems . . . 443

10.4 Input Handling in Window Systems 447
10.5 Interaction-Technique Toolkits 451
10.6 User-Interface Management Systems 456
Exercises 468

CHAPTER 11
REPRESENTING CURVES AND SURFACES 471

11.1 Polygon Meshes 473
11.2 Parametric Cubic Curves 478
11.3 Parametric Bicubic Surfaces 516
11.4 Quadric Surfaces 528
11.5 Summary 529
Exercises 530

CHAPTER 12
SOLID MODELING 533

12.1 Representing Solids 534
12.2 Regularized Boolean Set Operations 535
12.3 Primitive Instancing 539
12.4 Sweep Representations 540
12.5 Boundary Representations 542
12.6 Spatial-Partitioning Representations 548
12.7 Constructive Solid Geometry 557
12.8 Comparison of Representations 558
12.9 User Interfaces for Solid Modeling 561
12.10 Summary 561
Exercises 562

CHAPTER 13
ACHROMATIC AND COLORED LIGHT 563

13.1 Achromatic Light 563
13.2 Chromatic Color 574
13.3 Color Models for Raster Graphics 584
13.4 Reproducing Color 599
13.5 Using Color in Computer Graphics 601
13.6 Summary 603
Exercises 603

CHAPTER 14
THE QUEST FOR VISUAL REALISM 605

14.1 Why Realism? 606
14.2 Fundamental Difficulties 607
14.3 Rendering Techniques for Line Drawings 609

14.4 Rendering Techniques for Shaded Images 612
14.5 Improved Object Models 615
14.6 Dynamics 615
14.7 Stereopsis 616
14.8 Improved Displays 617
14.9 Interacting with Our Other Senses 617
14.10 Aliasing and Antialiasing 617
14.11 Summary 646
Exercises 647

CHAPTER 15
VISIBLE-SURFACE DETERMINATION 649

15.1 Functions of Two Variables 651
15.2 Techniques for Efficient Visible-Surface Algorithms 656
15.3 Algorithms for Visible-Line Determination 665
15.4 The z-Buffer Algorithm 668
15.5 List-Priority Algorithms 672
15.6 Scan-Line Algorithms 680
15.7 Area-Subdivision Algorithms 686
15.8 Algorithms for Octrees 695
15.9 Algorithms for Curved Surfaces 698
15.10 Visible-Surface Ray Tracing 701
15.11 Summary 715
Exercises 718

CHAPTER 16
ILLUMINATION AND SHADING 721

16.1 Illumination Models 722
16.2 Shading Models for Polygons 734
16.3 Surface Detail 741
16.4 Shadows 745
16.5 Transparency 754
16.6 Interobject Reflections 758
16.7 Physically Based Illumination Models 760
16.8 Extended Light Sources 772
16.9 Spectral Sampling 773
16.10 Improving the Camera Model 774
16.11 Global Illumination Algorithms 775
16.12 Recursive Ray Tracing 776
16.13 Radiosity Methods 793
16.14 The Rendering Pipeline 806
16.15 Summary 813
Exercises 813

CHAPTER 17
IMAGE MANIPULATION AND STORAGE 815

17.1 What Is an Image? . . . 816
17.2 Filtering . . . 817
17.3 Image Processing . . . 820
17.4 Geometric Transformations of Images . . . 820
17.5 Multipass Transformations . . . 828
17.6 Image Compositing . . . 835
17.7 Mechanisms for Image Storage . . . 843
17.8 Special Effects with Images . . . 850
17.9 Summary . . . 851
Exercises . . . 851

CHAPTER 18
ADVANCED RASTER GRAPHICS ARCHITECTURE 855

18.1 Simple Raster-Display System . . . 856
18.2 Display-Processor Systems . . . 861
18.3 Standard Graphics Pipeline . . . 866
18.4 Introduction to Multiprocessing . . . 873
18.5 Pipeline Front-End Architectures . . . 877
18.6 Parallel Front-End Architectures . . . 880
18.7 Multiprocessor Rasterization Architectures . . . 882
18.8 Image-Parallel Rasterization . . . 887
18.9 Object-Parallel Rasterization . . . 899
18.10 Hybrid-Parallel Rasterization . . . 902
18.11 Enhanced Display Capabilities . . . 907
18.12 Summary . . . 920
Exercises . . . 920

CHAPTER 19
ADVANCED GEOMETRIC AND RASTER ALGORITHMS 923

19.1 Clipping . . . 924
19.2 Scan-Converting Primitives . . . 945
19.3 Antialiasing . . . 965
19.4 The Special Problems of Text . . . 976
19.5 Filling Algorithms . . . 979
19.6 Making copyPixel Fast . . . 986
19.7 The Shape Data Structure and Shape Algebra . . . 992
19.8 Managing Windows with bitBlt . . . 996
19.9 Page-Description Languages . . . 998
19.10 Summary . . . 1006
Exercises . . . 1006

CHAPTER 20
ADVANCED MODELING TECHNIQUES 1011

20.1 Extensions of Previous Techniques 1012
20.2 Procedural Models 1018
20.3 Fractal Models 1020
20.4 Grammar-Based Models 1027
20.5 Particle Systems 1031
20.6 Volume Rendering 1034
20.7 Physically Based Modeling 1039
20.8 Special Models for Natural and Synthetic Objects 1043
20.9 Automating Object Placement 1050
20.10 Summary 1054
Exercises 1054

CHAPTER 21
ANIMATION 1057

21.1 Conventional and Computer-Assisted Animation 1058
21.2 Animation Languages 1065
21.3 Methods of Controlling Animation 1070
21.4 Basic Rules of Animation 1077
21.5 Problems Peculiar to Animation 1078
21.6 Summary 1080
Exercises 1080

APPENDIX: MATHEMATICS FOR COMPUTER GRAPHICS 1083

A.1 Vector Spaces and Affine Spaces 1083
A.2 Some Standard Constructions in Vector Spaces 1091
A.3 Dot Products and Distances 1094
A.4 Matrices 1103
A.5 Linear and Affine Transformations 1106
A.6 Eigenvalues and Eigenvectors 1108
A.7 Newton-Raphson Iteration for Root Finding 1109
Exercises 1111

BIBLIOGRAPHY 1113

INDEX 1153

1
Introduction

Computer graphics started with the display of data on hardcopy plotters and cathode ray tube (CRT) screens soon after the introduction of computers themselves. It has grown to include the creation, storage, and manipulation of models and images of objects. These models come from a diverse and expanding set of fields, and include physical, mathematical, engineering, architectural, and even conceptual (abstract) structures, natural phenomena, and so on. Computer graphics today is largely *interactive*: The user controls the contents, structure, and appearance of objects and of their displayed images by using input devices, such as a keyboard, mouse, or touch-sensitive panel on the screen. Because of the close relationship between the input devices and the display, the handling of such devices is included in the study of computer graphics.

Until the early 1980s, computer graphics was a small, specialized field, largely because the hardware was expensive and graphics-based application programs that were easy to use and cost-effective were few. Then, personal computers with built-in raster graphics displays—such as the Xerox Star and, later, the mass-produced, even less expensive Apple Macintosh and the IBM PC and its clones—popularized the use of *bitmap graphics* for user-computer interaction. A *bitmap* is a ones and zeros representation of the rectangular array of points (*pixels* or *pels*, short for "picture elements") on the screen. Once bitmap graphics became affordable, an explosion of easy-to-use and inexpensive graphics-based applications soon followed. Graphics-based user interfaces allowed millions of new users to control simple, low-cost application programs, such as spreadsheets, word processors, and drawing programs.

The concept of a "desktop" now became a popular metaphor for organizing screen space. By means of a *window manager*, the user could create, position, and resize

ATI Ex. 2003
IPR2023-00564
Page 14 of 241

rectangular screen areas, called *windows*, that acted as virtual graphics terminals, each running an application. This allowed users to switch among multiple activities just by pointing at the desired window, typically with the mouse. Like pieces of paper on a messy desk, windows could overlap arbitrarily. Also part of this desktop metaphor were displays of icons that represented not just data files and application programs, but also common office objects, such as file cabinets, mailboxes, printers, and trashcans, that performed the computer-operation equivalents of their real-life counterparts. *Direct manipulation* of objects via "pointing and clicking" replaced much of the typing of the arcane commands used in earlier operating systems and computer applications. Thus, users could select icons to activate the corresponding programs or objects, or select buttons on pull-down or pop-up screen menus to make choices. Today, almost all interactive application programs, even those for manipulating text (e.g., word processors) or numerical data (e.g., spreadsheet programs), use graphics extensively in the user interface and for visualizing and manipulating the application-specific objects. Graphical interaction via raster displays (displays using bitmaps) has replaced most textual interaction with alphanumeric terminals.

Even people who do not use computers in their daily work encounter computer graphics in television commercials and as cinematic special effects. Computer graphics is no longer a rarity. It is an integral part of all computer user interfaces, and is indispensable for visualizing two-dimensional (2D), three-dimensional (3D), and higher-dimensional objects. Areas as diverse as education, science, engineering, medicine, commerce, the military, advertising, and entertainment all rely on computer graphics. Learning how to program and use computers now includes learning how to use simple 2D graphics as a matter of routine.

1.1 IMAGE PROCESSING AS PICTURE ANALYSIS

Computer graphics concerns the pictorial *synthesis* of real or imaginary objects from their computer-based models, whereas the related field of *image processing* (also called *picture processing*) treats the converse process: the *analysis* of scenes, or the *reconstruction* of models of 2D or 3D objects from their pictures. Picture analysis is important in many arenas: aerial surveillance photographs, slow-scan television images of the moon or of planets gathered from space probes, television images taken from an industrial robot's "eye," chromosome scans, X-ray images, computerized axial tomography (CAT) scans, and fingerprint analysis all exploit image-processing technology (see Color Plate I.1). Image processing has the subareas *image enhancement, pattern detection and recognition*, and *scene analysis and computer vision*. Image enhancement deals with improving image quality by eliminating noise (extraneous or missing pixel data) or by enhancing contrast. Pattern detection and recognition deal with detecting and clarifying standard patterns and finding deviations (distortions) from these patterns. A particularly important example is optical character recognition (OCR) technology, which allows for the economical bulk input of pages of typeset, typewritten, or even handprinted characters. Scene analysis and computer vision allow scientists to recognize and reconstruct a 3D model of a scene from several 2D images. An example is an industrial robot sensing the relative sizes, shapes, positions, and colors of parts on a conveyor belt.

Although both computer graphics and image processing deal with computer processing of pictures, they have until recently been quite separate disciplines. Now that they both use raster displays, however, the overlap between the two is growing, as is particularly evident in two areas. First, in interactive image processing, human input via menus and other graphical interaction techniques helps to control various subprocesses while transformations of continuous-tone images are shown on the screen in real time. For example, scanned-in photographs are electronically touched up, cropped, and combined with others (even with synthetically generated images) before publication. Second, simple image-processing operations are often used in computer graphics to help synthesize the image of a model. Certain ways of transforming and combining synthetic images depend largely on image-processing operations.

1.2 THE ADVANTAGES OF INTERACTIVE GRAPHICS

Graphics provides one of the most natural means of communicating with a computer, since our highly developed 2D and 3D pattern-recognition abilities allow us to perceive and process pictorial data rapidly and efficiently. In many design, implementation, and construction processes today, the information pictures can give is virtually indispensable. Scientific visualization became an important field in the late 1980s, when scientists and engineers realized that they could not interpret the prodigious quantities of data produced in supercomputer runs without summarizing the data and highlighting trends and phenomena in various kinds of graphical representations.

Creating and reproducing pictures, however, presented technical problems that stood in the way of their widespread use. Thus, the ancient Chinese proverb "a picture is worth ten thousand words" became a cliché in our society only after the advent of inexpensive and simple technology for producing pictures—first the printing press, then photography.

Interactive computer graphics is the most important means of producing pictures since the invention of photography and television; it has the added advantage that, with the computer, we can make pictures not only of concrete, "real-world" objects but also of abstract, synthetic objects, such as mathematical surfaces in 4D (see Color Plates I.3 and I.4), and of data that have no inherent geometry, such as survey results. Furthermore, we are not confined to static images. Although static pictures are a good means of communicating information, dynamically varying pictures are frequently even better—to coin a phrase, a moving picture is worth ten thousand static ones. This is especially true for time-varying phenomena, both real (e.g., the deflection of an aircraft wing in supersonic flight, or the development of a human face from childhood through old age) and abstract (e.g., growth trends, such as nuclear energy use in the United States or population movement from cities to suburbs and back to the cities). Thus, a movie can show changes over time more graphically than can a sequence of slides. Similarly, a sequence of frames displayed on a screen at more than 15 frames per second can convey smooth motion or changing form better than can a jerky sequence, with several seconds between individual frames. The use of dynamics is especially effective when the user can control the animation by adjusting the speed, the portion of the total scene in view, the amount of detail shown, the geometric relationship of the objects in the scene to one another, and so on. Much of

interactive graphics technology therefore contains hardware and software for user-controlled motion dynamics and update dynamics.

With *motion dynamics*, objects can be moved and tumbled with respect to a stationary observer. The objects can also remain stationary and the viewer can move around them, pan to select the portion in view, and zoom in or out for more or less detail, as though looking through the viewfinder of a rapidly moving video camera. In many cases, both the objects and the camera are moving. A typical example is the flight simulator (Color Plates I.5a and I.5b), which combines a mechanical platform supporting a mock cockpit with display screens for windows. Computers control platform motion, gauges, and the simulated world of both stationary and moving objects through which the pilot navigates. These multimillion-dollar systems train pilots by letting the pilots maneuver a simulated craft over a simulated 3D landscape and around simulated vehicles. Much simpler flight simulators are among the most popular games on personal computers and workstations. Amusement parks also offer "motion-simulator" rides through simulated terrestrial and extraterrestrial landscapes. Video arcades offer graphics-based dexterity games (see Color Plate I.6) and racecar-driving simulators, video games exploiting interactive motion dynamics: The player can change speed and direction with the "gas pedal" and "steering wheel," as trees, buildings, and other cars go whizzing by (see Color Plate I.7). Similarly, motion dynamics lets the user fly around and through buildings, molecules, and 3D or 4D mathematical space. In another type of motion dynamics, the "camera" is held fixed, and the objects in the scene are moved relative to it. For example, a complex mechanical linkage, such as the linkage on a steam engine, can be animated by moving or rotating all the pieces appropriately.

Update dynamics is the actual change of the shape, color, or other properties of the objects being viewed. For instance, a system can display the deformations of an airplane structure in flight or the state changes in a block diagram of a nuclear reactor in response to the operator's manipulation of graphical representations of the many control mechanisms. The smoother the change, the more realistic and meaningful the result. Dynamic interactive graphics offers a large number of user-controllable modes with which to encode and communicate information: the 2D or 3D shape of objects in a picture, their gray scale or color, and the time variations of these properties. With the recent development of digital signal processing (DSP) and audio synthesis chips, audio feedback can now be provided to augment the graphical feedback and to make the simulated environment even more realistic.

Interactive computer graphics thus permits extensive, high-bandwidth user–computer interaction. This significantly enhances our ability to understand data, to perceive trends, and to visualize real or imaginary objects—indeed, to create "virtual worlds" that we can explore from arbitrary points of view (see Color Plates I.15 and I.16). By making communication more efficient, graphics makes possible higher-quality and more precise results or products, greater productivity, and lower analysis and design costs.

1.3 REPRESENTATIVE USES OF COMPUTER GRAPHICS

Computer graphics is used today in many different areas of industry, business, government, education, entertainment, and, most recently, the home. The list of applications is

enormous and is growing rapidly as computers with graphics capabilities become commodity products. Let's look at a representative sample of these areas.

- *User interfaces.* As we mentioned, most applications that run on personal computers and workstations, and even those that run on terminals attached to time-shared computers and network compute servers, have user interfaces that rely on desktop window systems to manage multiple simultaneous activities, and on point-and-click facilities to allow users to select menu items, icons, and objects on the screen; typing is necessary only to input text to be stored and manipulated. Word-processing, spreadsheet, and desktop-publishing programs are typical applications that take advantage of such user-interface techniques. The authors of this book used such programs to create both the text and the figures; then, the publisher and their contractors produced the book using similar typesetting and drawing software.

- *(Interactive) plotting in business, science, and technology.* The next most common use of graphics today is probably to create 2D and 3D graphs of mathematical, physical, and economic functions; histograms, bar and pie charts; task-scheduling charts; inventory and production charts; and the like. All these are used to present meaningfully and concisely the trends and patterns gleaned from data, so as to clarify complex phenomena and to facilitate informed decision making.

- *Office automation and electronic publishing.* The use of graphics for the creation and dissemination of information has increased enormously since the advent of desktop publishing on personal computers. Many organizations whose publications used to be printed by outside specialists can now produce printed materials inhouse. Office automation and electronic publishing can produce both traditional printed (hardcopy) documents and electronic (softcopy) documents that contain text, tables, graphs, and other forms of drawn or scanned-in graphics. Hypermedia systems that allow browsing of networks of interlinked multimedia documents are proliferating (see Color Plate I.2).

- *Computer-aided drafting and design.* In computer-aided design (CAD), interactive graphics is used to design components and systems of mechanical, electrical, electromechanical, and electronic devices, including structures such as buildings, automobile bodies, airplane and ship hulls, very large-scale-integrated (VLSI) chips, optical systems, and telephone and computer networks. Sometimes, the user merely wants to produce the precise drawings of components and assemblies, as for online drafting or architectural blueprints. Color Plate I.8 shows an example of such a 3D design program, intended for nonprofessionals, a "customize your own patio deck" program used in lumber yards. More frequently, however, the emphasis is on interacting with a computer-based model of the component or system being designed in order to test, for example, its structural, electrical, or thermal properties. Often, the model is interpreted by a simulator that feeds back the behavior of the system to the user for further interactive design and test cycles. After objects have been designed, utility programs can *postprocess* the design database to make parts lists, to process "bills of materials," to define numerical control tapes for cutting or drilling parts, and so on.

- *Simulation and animation for scientific visualization and entertainment.* Computer-produced animated movies and displays of the time-varying behavior of real and simulated

objects are becoming increasingly popular for scientific and engineering visualization (see Color Plate I.10). We can use them to study abstract mathematical entities as well as mathematical models of such phenomena as fluid flow, relativity, nuclear and chemical reactions, physiological system and organ function, and deformation of mechanical structures under various kinds of loads. Another advanced-technology area is interactive cartooning. The simpler kinds of systems for producing "flat" cartoons are becoming cost-effective in creating routine "in-between" frames that interpolate between two explicitly specified "key frames." Cartoon characters will increasingly be modeled in the computer as 3D shape descriptions whose movements are controlled by computer commands, rather than by the figures being drawn manually by cartoonists (see Color Plates D and F). Television commercials featuring flying logos and more exotic visual trickery have become common, as have elegant special effects in movies (see Color Plates I.12, I.13, II.18, and G). Sophisticated mechanisms are available to model the objects and to represent light and shadows.

- *Art and commerce*. Overlapping the previous category is the use of computer graphics in art and advertising; here, computer graphics is used to produce pictures that express a message and attract attention (see Color Plates I.9, I.11, and H). Personal computers and Teletext and Videotex terminals in public places such as museums, transportation terminals, supermarkets, and hotels, as well as in private homes, offer much simpler but still informative pictures that let users orient themselves, make choices, or even "teleshop" and conduct other business transactions. Finally, slide production for commercial, scientific, or educational presentations is another cost-effective use of graphics, given the steeply rising labor costs of the traditional means of creating such material.

- *Process control*. Whereas flight simulators or arcade games let users interact with a simulation of a real or artificial world, many other applications enable people to interact with some aspect of the real world itself. Status displays in refineries, power plants, and computer networks show data values from sensors attached to critical system components, so that operators can respond to problematic conditions. For example, military commanders view field data—number and position of vehicles, weapons launched, troop movements, casualties—on *command and control* displays to revise their tactics as needed; flight controllers at airports see computer-generated identification and status information for the aircraft blips on their radar scopes, and can thus control traffic more quickly and accurately than they could with the unannotated radar data alone; spacecraft controllers monitor telemetry data and take corrective action as needed.

- *Cartography*. Computer graphics is used to produce both accurate and schematic representations of geographical and other natural phenomena from measurement data. Examples include geographic maps, relief maps, exploration maps for drilling and mining, oceanographic charts, weather maps, contour maps, and population-density maps.

1.4 CLASSIFICATION OF APPLICATIONS

The diverse uses of computer graphics listed in the previous section differ in a variety of ways, and a number of classifications may be used to categorize them. The first

classification is by *type (dimensionality) of the object* to be represented and the *kind of picture* to be produced. The range of possible combinations is indicated in Table 1.1.

Some of the objects represented graphically are clearly abstract, some are real; similarly, the pictures can be purely symbolic (a simple 2D graph) or realistic (a rendition of a still life). The same object can, of course, be represented in a variety of ways. For example, an electronic printed circuit board populated with integrated circuits can be portrayed by many different 2D symbolic representations or by 3D synthetic photographs of the board.

The second classification is by the *type of interaction*, which determines the user's degree of control over the object and its image. The range here includes *offline plotting*, with a predefined database produced by other application programs or digitized from physical models, *interactive plotting*, in which the user controls iterations of "supply some parameters, plot, alter parameters, replot", *predefining or calculating the object and flying around it* in real time under user control, as in real-time animation systems used for scientific visualization and flight simulators; and *interactive designing*, in which the user starts with a blank screen, defines new objects (typically by assembling them from predefined components), and then moves around to get a desired view.

The third classification is by the *role of the picture*, or the degree to which the picture is an end in itself or is merely a means to an end. In cartography, drafting, raster painting, animation, and artwork, for example, the drawing is the end product; in many CAD applications, however, the drawing is merely a representation of the geometric properties of the object being designed or analyzed. Here the drawing or construction phase is an important but small part of a larger process, the goal of which is to create and postprocess a common database using an integrated suite of application programs.

A good example of graphics in CAD is the creation of a VLSI chip. The engineer makes a preliminary chip design using a CAD package. Once all the gates are laid out, she then subjects the chip to hours of simulated use. From the first run, for instance, she learns that the chip works only at clock speeds above 80 nanoseconds (ns). Since the target clock speed of the machine is 50 ns, the engineer calls up the initial layout and redesigns a portion of the logic to reduce its number of stages. On the second simulation run, she learns that the chip will not work at speeds below 60 ns. Once again, she calls up the drawing and redesigns a portion of the chip. Once the chip passes all the simulation tests, she invokes a postprocessor to create a database of information for the manufacturer about design and materials specifications, such as conductor path routing and assembly drawings. In this

TABLE 1.1 CLASSIFICATION OF COMPUTER GRAPHICS BY OBJECT AND PICTURE

Type of object	Pictorial representation	Example
2D	Line drawing	Fig. 2.1
	Gray scale image	Fig. 1.1
	Color image	Color Plate I.2
3D	Line drawing (or *wireframe*)	Color Plates II.21–II.23
	Line drawing, with various effects	Color Plates II.24–II.27
	Shaded, color image with various effects	Color Plates II.28–II.39

example, the representation of the chip's geometry produces output beyond the picture itself. In fact, the geometry shown on the screen may contain *less* detail than the underlying database.

A final categorization arises from the logical and temporal *relationship between objects and their pictures*. The user may deal, for example, with only one picture at a time (typical in plotting), with a time-varying sequence of related pictures (as in motion or update dynamics), or with a structured collection of objects (as in many CAD applications that contain hierarchies of assembly and subassembly drawings).

1.5 DEVELOPMENT OF HARDWARE AND SOFTWARE FOR COMPUTER GRAPHICS

This book concentrates on fundamental principles and techniques that were derived in the past and are still applicable today—and generally will be applicable in the future. In this section, we take a brief look at the historical development of computer graphics, to place today's systems in context. Fuller treatments of the interesting evolution of this field are presented in [PRIN71], [MACH78], [CHAS81],[4] and [CACM84]. It is easier to chronicle the evolution of hardware than to document that of software, since hardware evolution has had a greater influence on how the field developed. Thus, we begin with hardware.

Crude plotting on hardcopy devices such as teletypes and line printers dates from the early days of computing. The Whirlwind Computer developed in 1950 at the Massachusetts Institute of Technology (MIT) had computer-driven CRT displays for output, both for operator use and for cameras producing hardcopy. The SAGE air-defense system developed in the middle 1950s was the first to use *command and control* CRT display consoles on which operators identified targets with light pens (hand-held pointing devices that sense light emitted by objects on the screen). The beginnings of modern interactive graphics, however, are found in Ivan Sutherland's seminal doctoral work on the Sketchpad drawing system [SUTH63]. He introduced data structures for storing symbol hierarchies built up via easy replication of standard components, a technique akin to the use of plastic templates for drawing circuit symbols. He also developed interaction techniques that used the keyboard and light pen for making choices, pointing, and drawing, and formulated many other fundamental ideas and techniques still in use today. Indeed, many of the features introduced in Sketchpad are found in the PHIGS graphics package discussed in Chapter 7.

At the same time, it was becoming clear to computer, automobile, and aerospace manufacturers that CAD and computer-aided manufacturing (CAM) activities had enormous potential for automating drafting and other drawing-intensive activities. The General Motors DAC system [JACK64] for automobile design, and the Itek Digitek system [CHAS81] for lens design, were pioneering efforts that showed the utility of graphical interaction in the iterative design cycles common in engineering. By the mid-sixties, a number of research projects and commercial products had appeared.

Since at that time computer input/output (I/O) was done primarily in batch mode using punched cards, hopes were high for a breakthrough in interactive user–computer communication. Interactive graphics, as "the window on the computer," was to be an integral part of vastly accelerated interactive design cycles. The results were not nearly so dramatic,

however, since interactive graphics remained beyond the resources of all but the most technology-intensive organizations. Among the reasons for this were these:

- The high *cost* of the graphics hardware, when produced without benefit of economies of scale—at a time when automobiles cost a few thousand dollars, computers cost several millions of dollars, and the first commercial computer displays cost more than a hundred thousand dollars.
- The need for large-scale, expensive *computing resources* to support massive design databases, interactive picture manipulation, and the typically large suite of postprocessing programs whose input came from the graphics-design phase.
- The *difficulty of writing large, interactive programs* for the new time-sharing environment at a time when both graphics and interaction were new to predominantly batch-oriented FORTRAN programmers.
- *One-of-a-kind, nonportable software*, typically written for a particular manufacturer's display device and produced without the benefit of modern software-engineering principles for building modular, structured systems; when software is nonportable, moving to new display devices necessitates expensive and time-consuming rewriting of working programs.

It was the advent of graphics-based personal computers, such as the Apple Macintosh and the IBM PC, that finally drove down the costs of both hardware and software so dramatically that millions of graphics computers were sold as "appliances" for office and home; when the field started in the early sixties, its practitioners never dreamed that personal computers featuring graphical interaction would become so common so soon.

1.5.1 Output Technology

The display devices developed in the mid-sixties and in common use until the mid-eighties are called *vector, stroke, line drawing, or calligraphic displays*. The term *vector* is used as a synonym for *line* here; a *stroke* is a short line, and *characters* are made of sequences of such strokes. We shall look briefly at vector-system architecture, because many modern raster graphics systems use similar techniques. A typical vector system consists of a display processor connected as an I/O peripheral to the central processing unit (CPU), a display buffer memory, and a CRT. The buffer stores the computer-produced *display list* or *display program*; it contains point- and line-plotting commands with (x, y) or (x, y, z) endpoint coordinates, as well as character-plotting commands. Figure 1.4 shows a typical vector architecture; the display list in memory is shown as a symbolic representation of the output commands and their (x, y) or character values.

The commands for plotting points, lines, and characters are interpreted by the display processor. It sends digital and point coordinates to a *vector generator* that converts the digital coordinate values to analog voltages for beam-deflection circuits that displace an electron beam writing on the CRT's phosphor coating (the details are given in Chapter 4). The essence of a vector system is that the beam is deflected from endpoint to endpoint, as dictated by the arbitrary order of the display commands; this technique is called *random scan*. (Laser shows also use random-scan deflection of the laser beam.) Since the light output of the phosphor decays in tens or at most hundreds of microseconds, the display

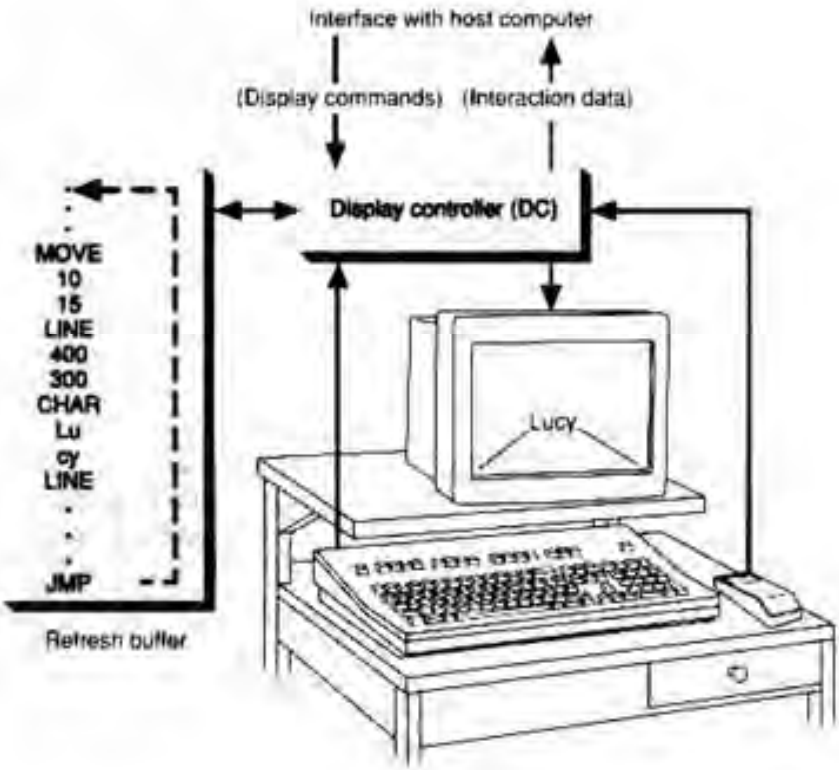


Fig. 1.1 Architecture of a vector display.

processor must cycle through the display list to *refresh* the phosphor at least 30 times per second (30 Hz) to avoid flicker; hence, the buffer holding the display list is usually called a *refresh buffer*. Note that, in Fig. 1.1, the jump instruction loops back to the top of the display list to provide the cyclic refresh.

In the sixties, buffer memory and processors fast enough to refresh at (at least) 30 Hz were expensive, and only a few thousand lines could be shown without noticeable flicker. Then, in the late sixties, the direct-view storage tube (DVST) obviated both the buffer and the refresh process, and eliminated all flicker. This was the vital step in making interactive graphics affordable. A DVST stores an image by writing that image once with a relatively slow-moving electron beam on a storage mesh in which the phosphor is embedded. The small, self-sufficient DVST terminal was an order of magnitude less expensive than was the typical refresh system; further, it was ideal for a low-speed (300- to1200-baud) telephone interface to time-sharing systems. DVST terminals introduced many users and programmers to interactive graphics.

Another major hardware advance of the late sixties was attaching the display to a minicomputer; with this configuration, the central time-sharing computer was relieved of the heavy demands of refreshed display devices, especially user-interaction handling, and updating the image on the screen. The minicomputer typically ran application programs as

well, and could in turn be connected to the larger central mainframe to run large analysis programs. Both minicomputer and DVST configurations led to installations of thousands of graphics systems. Also at this time, the hardware of the display processor itself was becoming more sophisticated, taking over many routine but time-consuming jobs from the graphics software. Foremost among such devices was the invention in 1968 of refresh display hardware for geometric transformations that could scale, rotate, and translate points and lines on the screen in real time, could perform 2D and 3D clipping, and could produce parallel and perspective projections (see Chapter 6).

The development in the early seventies of inexpensive raster graphics, based on television technology, contributed more to the growth of the field than did any other technology. *Raster displays* store the display *primitives* (such as lines, characters, and solidly shaded or patterned areas) in a refresh buffer in terms of their component pixels, as shown in Fig. 1.2. In some raster displays, there is a hardware display controller that receives and interprets sequences of output commands similar to those of the vector displays (as shown in the figure); in simpler, more common systems, such as those in personal computers, the display controller exists only as a software component of the graphics library package, and the refresh buffer is just a piece of the CPU's memory that can be read out by the image display subsystem (often called the video controller) that produces the actual image on the screen.

The complete image on a raster display is formed from the *raster*, which is a set of horizontal *raster lines*, each a row of individual pixels; the raster is thus stored as a matrix of pixels representing the entire screen area. The entire image is scanned out sequentially by

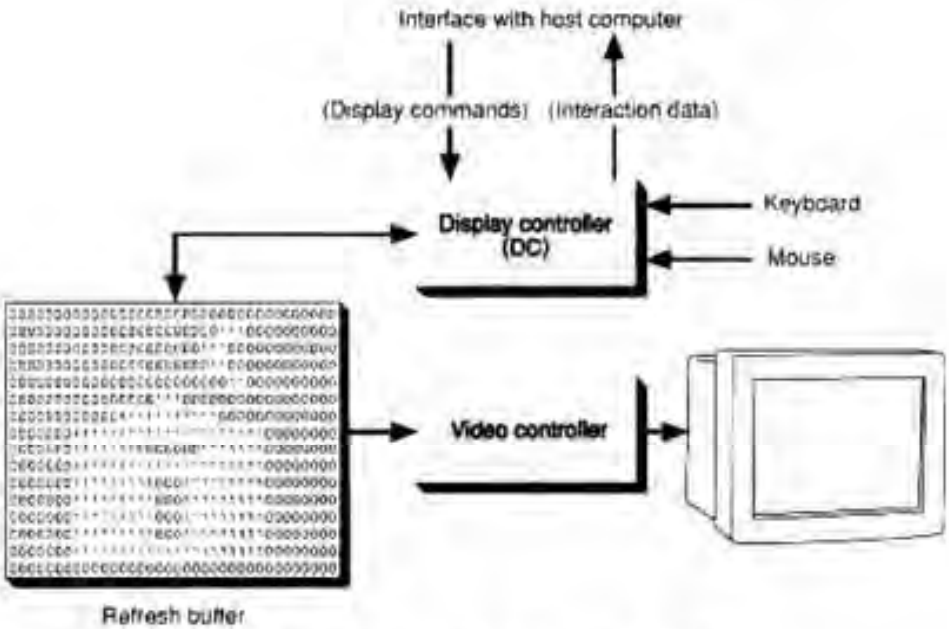


Fig. 1.2 Architecture of a raster display

the video controller, one raster line at a time, from top to bottom and then back to the top (as shown in Fig. 1.3). At each pixel, the beam's intensity is set to reflect the pixel's intensity; in color systems, three beams are controlled—one each for the red, green, and blue primary colors—as specified by the three color components of each pixel's value (see Chapters 4 and 13). Figure 1.4 shows the difference between random and raster scan for displaying a simple 2D line drawing of a house (part a). In part (b), the vector arcs are notated with arrowheads showing the random deflection of the beam. Dotted lines denote deflection of the beam, which is not turned on ("blanked"), so that no vector is drawn. Part (c) shows the unfilled house rendered by rectangles, polygons, and arcs, whereas part (d) shows a filled version. Note the jagged appearance of the lines and arcs in the raster scan images of parts (c) and (d); we shall discuss that visual artifact shortly.

In the early days of raster graphics, refreshing was done at television rates of 30 Hz; today, a 60-Hz or higher refresh rate is used to avoid flickering of the image. Whereas in a vector system the refresh buffer stored op-codes and endpoint coordinate values, in a raster system the entire image of, say, 1024 lines of 1024 pixels each, must be stored explicitly. The term *bitmap* is still in common use to describe both the refresh buffer and the array of pixel values that map one for one to pixels on the screen. Bitmap graphics has the advantage over vector graphics that the actual display of the image is handled by inexpensive scan-out logic. The regular, repetitive raster scan is far easier and less expensive to implement than is the random scan of vector systems, whose vector generators must be highly accurate to provide linearity and repeatability of the beam's deflection.

The availability of inexpensive solid-state random-access memory (RAM) for bitmaps in the early seventies was the breakthrough needed to make raster graphics the dominant hardware technology. Bilevel (also called monochrome) CRTs draw images in black and white or black and green; some plasma panels use orange and black. Bilevel bitmaps contain a single bit per pixel, and the entire bitmap for a screen with a resolution of 1024 by 1024 pixels is only 2^{20} bits, or about 128,000 bytes. Low-end color systems have 8 bits per pixel, allowing 256 colors simultaneously; more expensive systems have 24 bits per pixel,

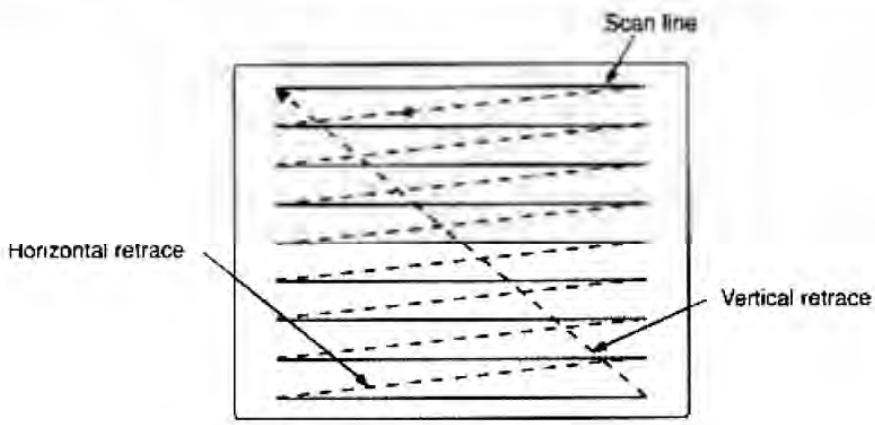


Fig. 1.3 Raster scan.

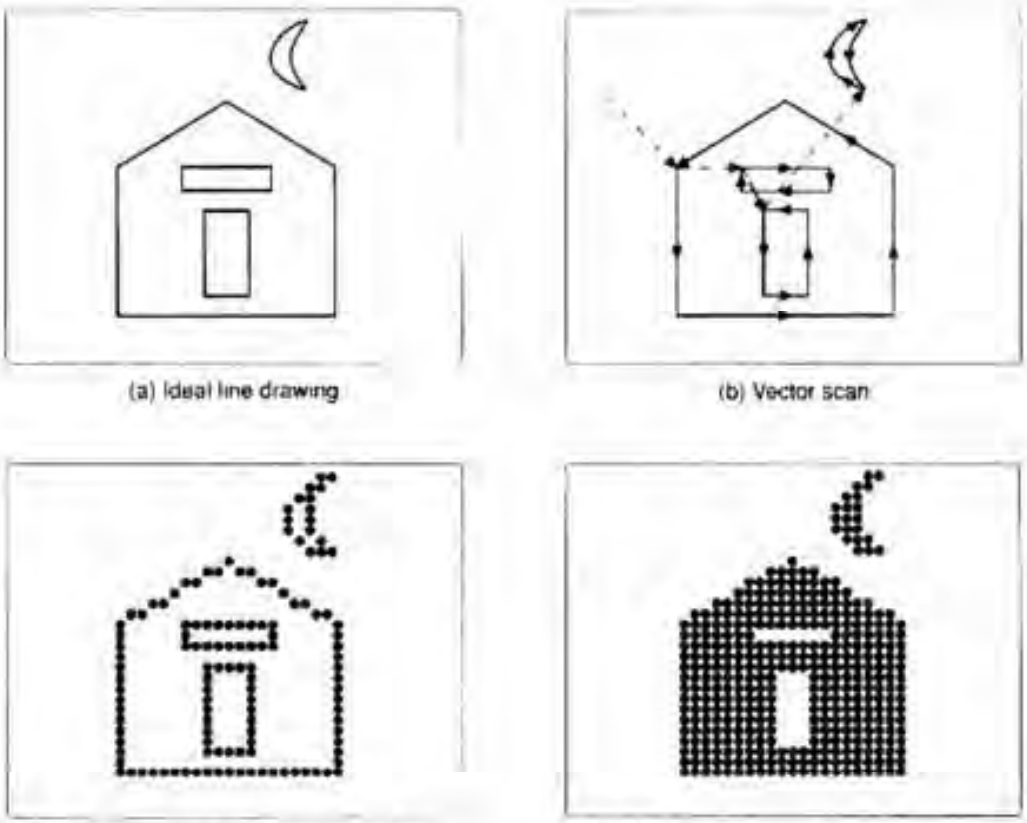


Fig. 1.4 Random scan versus raster scan. We symbolize the screen as a rounded rectangle filled with a light gray shade that denotes the white background; the image is drawn in black on this background.

allowing a choice of any of 16 million colors; and refresh buffers with 32 bits per pixel, and screen resolution of 1280 by 1024 pixels are available even on personal computers. Of the 32 bits, 24 bits are devoted to representing color, and 8 to control purposes, as discussed in Chapter 4. Beyond that, buffers with 96 bits (or more) per pixel[1] are available at 1280 by 1024 resolution on the high-end systems discussed in Chapter 18. A typical 1280 by 1024 color system with 24 bits per pixel requires 3.75 MB of RAM—inexpensive by today's standards. The term *bitmap*, strictly speaking, applies only to 1-bit-per-pixel bilevel systems; for multiple-bit-per-pixel systems, we use the more general term *pixmap* (short for pixel map). Since *pixmap* in common parlance refers both to the contents of the refresh

[1]Of these 96 bits, typically 64 bits are used for two 32-bit color-and-control buffers to allow *double-buffering* of two images: while one image is being refreshed, the second one is being updated. The remaining 32-bit buffer is used to implement a hardware technique called *z*-buffering, used to do visible-surface determination for creating realistic 3D images (see Chapters 14 and 15).

buffer and to the buffer memory itself, we use the term *frame buffer* when we mean the actual buffer memory.

The major advantages of raster graphics over vector graphics include lower cost and the ability to display areas filled with solid colors or patterns, an especially rich means of communicating information that is essential for realistic images of 3D objects. Furthermore, the refresh process is independent of the complexity (number of polygons, etc.) of the image, since the hardware is fast enough that each pixel in the buffer can be read out on each refresh cycle. Most people do not perceive flicker on screens refreshed above 70 Hz. In contrast, vector displays flicker when the number of primitives in the buffer becomes too large; typically a maximum of a few hundred thousand short vectors may be displayed flicker-free.

The major disadvantage of raster systems compared to vector systems arises from the discrete nature of the pixel representation. First, primitives such as lines and polygons are specified in terms of their endpoints (vertices) and must be *scan-converted* into their component pixels in the frame buffer. The term derives from the notion that the programmer specifies endpoint or vertex coordinates in random-scan mode, and this information must be reduced by the system to pixels for raster-scan–mode display. Scan conversion is commonly done with software in personal computers and low-end workstations, where the microprocessor CPU is responsible for all graphics. For higher performance, scan conversion can be done by special-purpose hardware, including *raster image processor* (RIP) chips used as coprocessors or accelerators.

Because each primitive must be scan-converted, real-time dynamics is far more computationally demanding on raster systems than on vector systems. First, transforming 1000 lines on a vector system can mean transforming 2000 endpoints in the worst case. In the next refresh cycle, the vector-generator hardware automatically redraws the transformed lines in their new positions. In a raster system, however, not only must the endpoints be transformed (using hardware transformation units identical to those used by vector systems), but also each transformed primitive must then be scan-converted using its new endpoints, which define its new size and position. None of the contents of the frame buffer can be salvaged. When the CPU is responsible for both endpoint transformation and scan conversion, only a small number of primitives can be transformed in real time. Transformation and scan-conversion hardware is thus needed for dynamics in raster systems; as a result of steady progress in VLSI, that has become feasible even in low-end systems.

The second drawback of raster systems arises from the nature of the raster itself. Whereas a vector system can draw a continuous, smooth line (and even some smooth curves) from essentially any point on the CRT face to any other, the raster system can display mathematically smooth lines, polygons, and boundaries of curved primitives such as circles and ellipses only by approximating them with pixels on the raster grid. This can cause the familiar problem of "jaggies" or "staircasing," as shown in Fig. 1.4 (c) and (d). This visual artifact is a manifestation of a sampling error called *aliasing* in signal-processing theory; such artifacts occur when a function of a continuous variable that contains sharp changes in intensity is approximated with discrete samples. Both theory and practice in modern computer graphics are concerned with techniques for *antialiasing* on

gray-scale or color systems. These techniques specify gradations in intensity of neighboring pixels at edges of primitives, rather than setting pixels to maximum or zero intensity only; see Chapters 3, 14, and 19 for further discussion of this important topic.

1.5.2 Input Technology

Input technology has also improved greatly over the years. The clumsy, fragile light pen of vector systems has been replaced by the ubiquitous mouse (first developed by office-automation pioneer Doug Engelbart in the mid-sixties [ENGE68]), the data tablet, and the transparent, touch-sensitive panel mounted on the screen. Even fancier input devices that supply not just (x, y) locations on the screen, but also 3D and even higher-dimensional input values (degrees of freedom), are becoming common, as discussed in Chapter 8. Audio communication also has exciting potential, since it allows hands-free input and natural output of simple instructions, feedback, and so on. With the standard input devices, the user can specify operations or picture components by typing or drawing new information or by pointing to existing information on the screen. These interactions require no knowledge of programming and only a little keyboard use: The user makes choices simply by selecting menu buttons or icons, answers questions by checking options or typing a few characters in a form, places copies of predefined symbols on the screen, draws by indicating consecutive endpoints to be connected by straight lines or interpolated by smooth curves, paints by moving the cursor over the screen, and fills closed areas bounded by polygons or paint contours with shades of gray, colors, or various patterns.

1.5.3 Software Portability and Graphics Standards

Steady advances in hardware technology have thus made possible the evolution of graphics displays from one-of-a-kind special output devices to the standard human interface to the computer. We may well wonder whether software has kept pace. For example, to what extent have early difficulties with overly complex, cumbersome, and expensive graphics systems and application software been resolved? Many of these difficulties arose from the primitive graphics software that was available, and in general there has been a long, slow process of maturation in such software. We have moved from low-level, *device-dependent* packages supplied by manufacturers for their particular display devices to higher-level, *device-independent* packages. These packages can drive a wide variety of display devices, from laser printers and plotters to film recorders and high-performance real-time displays. The main purpose of using a device-independent package in conjunction with a high-level programming language is to promote *application-program portability*. This portability is provided in much the same way as a high-level, machine-independent language (such as FORTRAN, Pascal, or C) provides portability: by isolating the programmer from most machine peculiarities and providing language features readily implemented on a broad range of processors. "Programmer portability" is also enhanced in that programmers can now move from system to system, or even from installation to installation, and find familiar software.

A general awareness of the need for standards in such device-independent graphics packages arose in the mid-seventies and culminated in a specification for a *3D Core*

Graphics System (the Core, for short) produced by an ACM SIGGRAPH[2] Committee in 1977 [GSPC77] and refined in 1979 [GSPC79]. The first three authors of this book were actively involved in the design and documentation of the 1977 Core.

The Core specification fulfilled its intended role as a baseline specification. Not only did it have many implementations, but also it was used as input to official (governmental) standards projects within both ANSI (the American National Standards Institute) and ISO (the International Standards Organization). The first graphics specification to be officially standardized was GKS (the Graphical Kernel System [ANSI85b]), an elaborated, cleaned-up version of the Core that, unlike the Core, was restricted to 2D. In 1988, GKS-3D [INTE88], a 3D extension of GKS, became an official standard, as did a much more sophisticated but even more complex graphics system called PHIGS (Programmer's Hierarchical Interactive Graphics System [ANSI88]). GKS supports the grouping of logically related primitives—such as lines, polygons, and character strings—and their attributes into collections called *segments*; these segments may not be nested. PHIGS, as its name implies, does support nested hierarchical groupings of 3D primitives, called *structures*. In PHIGS, all primitives, including invocations of substructures, are subject to geometric transformations (scaling, rotation, and translation) to accomplish dynamic movement. PHIGS also supports a retained database of structures that the programmer may edit selectively; PHIGS automatically updates the screen whenever the database has been altered. PHIGS has been extended with a set of features for modern, pseudorealistic rendering[3] of objects on raster displays; this extension is called PHIGS+ [PHIG88]. PHIGS implementations are large packages, due to the many features and to the complexity of the specification. PHIGS and especially PHIGS+ implementations run best when there is hardware support for their transformation, clipping, and rendering features.

This book discusses graphics software standards at some length. We first study SRGP (the Simple Raster Graphics Package), which borrows features from Apple's popular QuickDraw integer raster graphics package [ROSE85] and MIT's X Window System [SCHE88a] for output and from GKS and PHIGS for input. Having looked at simple applications in this low-level raster graphics package, we then study the scan-conversion and clipping algorithms such packages use to generate images of primitives in the frame buffer. Then, after building a mathematical foundation for 2D and 3D geometric transformations and for parallel and perspective viewing in 3D, we study a far more powerful package called SPHIGS (Simple PHIGS). SPHIGS is a subset of PHIGS that operates on primitives defined in a floating-point, abstract, 3D world-coordinate system

[2]SIGGRAPH is the Special Interest Group on Graphics, one of the professional groups within ACM, the Association for Computing Machinery. ACM is one of the two major professional societies for computer professionals; the IEEE Computer Society is the other. SIGGRAPH publishes a research journal and sponsors an annual conference that features presentations of research papers in the field and an equipment exhibition. The Computer Society also publishes a research journal in graphics.

[3]A pseudorealistic rendering is one that simulates the simple laws of optics describing how light is reflected by objects. Photorealistic rendering uses better approximations to the way objects reflect and refract light; these approximations require more computation but produce images that are more nearly photographic in quality (see Color Plate E).

independent of any type of display technology, and that supports some simple PHIGS+ features. We have oriented our discussion to PHIGS and PHIGS+ because we believe they will have much more influence on interactive 3D graphics than will GKS-3D, especially given the increasing availability of hardware that supports real-time transformations and rendering of pseudorealistic images.

1.6 CONCEPTUAL FRAMEWORK FOR INTERACTIVE GRAPHICS

1.6.1 Overview

The high-level conceptual framework shown in Fig. 1.5 can be used to describe almost any interactive graphics system. At the hardware level (not shown explicitly in the diagram), a computer receives input from interaction devices, and outputs images to a display device. The software has three components. The first is the *application program*; it creates, stores into, and retrieves from the second component, the *application model*, which represents the data or objects to be pictured on the screen. The application program also handles user input. It produces views by sending to the third component, the *graphics system*, a series of graphics output commands that contain both a detailed geometric description of *what* is to be viewed and the attributes describing *how* the objects should appear. The graphics system is responsible for actually producing the picture from the detailed descriptions and for passing the user's input to the application program for processing.

The graphics system is thus an intermediary between the application program and the display hardware that effects an *output transformation* from objects in the application model to a view of the model. Symmetrically, it effects an *input transformation* from user actions to inputs to the application program that will cause the application to make changes in the model and/or picture. The fundamental task of the designer of an interactive graphics application program is to specify what classes of data items or objects are to be generated and represented pictorially, and how the user and the application program are to interact in order to create and modify the model and its visual representation. Most of the programmer's task concerns creating and editing the model and handling user interaction, not actually creating views, since that is handled by the graphics system.

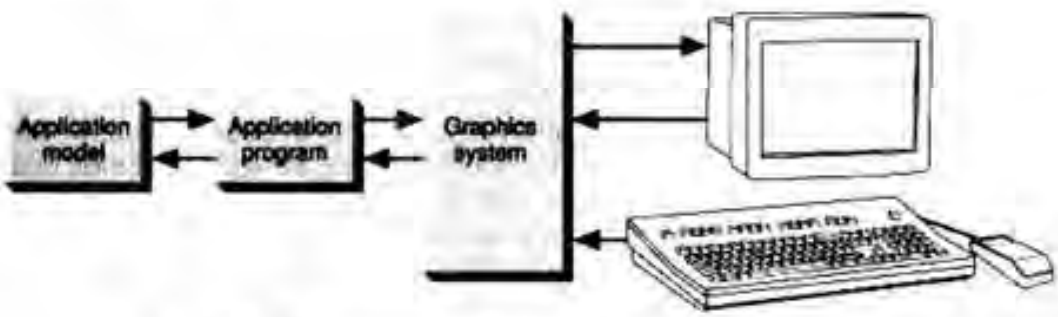


Fig. 1.5 Conceptual framework for interactive graphics.

1.6.2 Application Modeling

The application model captures all the data, objects, and relationships among them that are relevant to the display and interaction part of the application program and to any nongraphical postprocessing modules. Examples of such postprocessing modules are analyses of the transient behavior of a circuit or of the stresses in an aircraft wing, simulation of a population model or a weather system, and pricing computations for a building. In the class of applications typified by "painting" programs such as MacPaint and PCPaint, the intent of the program is to produce an image by letting the user set or modify individual pixels. Here an explicit application model is not needed—the picture is both means and end, and the displayed bitmap or pixmap serves in effect as the application model.

More typically, however, there is an identifiable application model representing application objects through some combination of data and procedural description that is independent of a particular display device. Procedural descriptions are used, for example, to define fractals, as described in Section 20.3. A data model can be as rudimentary as an array of data points or as complex as a linked list representing a network data structure or a relational database storing a set of relations. We often speak of storing the *application model* in the *application database*; the terms are used interchangeably here. Models typically store descriptions of *primitives* (points, lines and polygons in 2D or 3D, and polyhedra and free-form surfaces in 3D) that define the shape of components of the object, object *attributes* such as line style, color, or surface texture; and *connectivity* relationships and positioning data that describe how the components fit together.

The objects stored in the model can differ greatly in the amount of intrinsic geometry needed to specify them. At the geometry-is-everything end of the spectrum, an industrial robot of the type discussed in Chapter 7 is described almost completely in terms of the geometry of its component polyhedra, each of which is a collection of 3D polygonal facets connected at common edges defined in terms of common vertices and enclosing a volume. A spreadsheet has much less intrinsic geometry. The spatial relationships between adjacent cells are stored, but the exact size or placement of each cell on the "paper" is not stored; instead, these values are determined dynamically by the spreadsheet program as a function of the contents of cells. At the geometry-free end of the spectrum, a demographic model storing statistics, such as income and age of individuals in some population, has no intrinsic geometry. These statistics can then be operated on by a procedure to derive some *geometrical interpretation*, such as a 2D graph, scatter diagram, or histogram.

Another class of applications without intrinsic geometry deals with the directed-graph networks used in various fields such as engineering and project management. These networks may be represented internally by adjacency matrices describing how nodes are connected, plus some property data for nodes and edges, and the application must then derive a layout in a predefined format in order to create a view of the graph. This representation might be created once and subsequently edited, as for a VLSI circuit layout computed over many hours. The model would then contain both a nongeometric and an almost purely geometric description of the circuit. Alternatively, if a layout for a particular application is simple and fast enough to derive, such as a project-scheduling chart with

labeled boxes and arrows, it can be created on the fly each time the data on which it is based change.

Geometric data in the application model often are accompanied by nongeometric textual or numeric property information useful to a postprocessing program or the interactive user. Examples of such data in CAD applications include manufacturing data; "price and supplier" data; thermal, mechanical, electrical, or electronic properties; and mechanical or electrical tolerances.

1.6.3 Describing to the Graphics System What Is to Be Viewed

The application program creates the application model either a priori as a result of prior computation, as in an engineering or scientific simulation on a supercomputer, or as part of an interactive session at the display device during which the user guides the construction process step by step to choose components and geometric and nongeometric property data. The user can ask the application program at any time to show a view of the model it has created so far. (The word *view* is used intentionally here, both in the sense of a visual rendering of some geometric properties of the objects being modeled and in the technical database sense of a 2D presentation of some properties of some subset of the model.)

Models are application-specific and are created independently of any particular display system. Therefore, the application program must convert a description of the portion of the model to be viewed from the internal representation of the geometry (whether explicitly stored in the model or derived on the fly) to whatever procedure calls or commands the graphics system uses to create an image. This conversion process has two phases. First, the application program traverses the application database that stores the model in order to extract the portions to be viewed, using some selection or query criteria. Then, the extracted geometry is put in a format that can be sent to the graphics system. The selection criteria can be geometric in nature (e.g., the portion of the model to be viewed has been shifted via the graphics equivalent of a pan or zoom camera operation), or they can be similar to traditional database query criteria (e.g., create a view of all the activities after March 15, 1989 in the scheduling chart for producing this book).

The data extracted during the database traversal must either be geometric in nature or must be converted to geometric data. The data can be described to the graphics system in terms of both primitives that the system can display directly and attributes that control the primitives' appearance. Display primitives typically match those stored in geometric models: lines, rectangles, polygons, circles, ellipses, and text in 2D, and polygons, polyhedra, and text in 3D. Advanced graphics systems such as PHIGS+ support additional primitives, including curves and surfaces defined by polynomials of higher degrees.

If the model stores geometric primitives not directly supported by the graphics package, the application program must reduce them to those that are accepted by the system. For example, if spheres and free-form surfaces are not supported, the application must approximate such surfaces by *tiling* or *tessellating* them with meshes of polygons that the graphics system can handle. Appearance attributes supported by the graphics package also tend to correspond to the ones stored in the model, such as color, line style, and line width. In addition to primitives and attributes, advanced graphics packages such as PHIGS

support facilities for specifying geometric transformations for scaling, rotating, and positioning components, for specifying how components are to be viewed in 3D, and for grouping logically related primitives, attributes, and transformations so that they can be invoked by a single reference to a named structure.

The graphics system typically consists of a set of output subroutines corresponding to the various primitives, attributes, and other elements. These are collected in a *graphics-subroutine library* or *package* that can be called from high-level languages such as C, Pascal, or LISP. The application program specifies geometric primitives and attributes to these subroutines, and the subroutines then drive the specific display device and cause it to display the image. Much as conventional I/O systems create logical I/O units to shield the application programmer from the messy details of hardware and device drivers, graphics systems create a *logical display device*. Thus, the graphics programmer can ignore such details as which part of image generation is done in the display hardware and which is done by the graphics package, or what the coordinate system of the display is. This abstraction of the display device pertains both to the output of images and to interaction via logical input devices. For example, the mouse, data tablet, touch panel, 2D joystick, or trackball can all be treated as the *locator* logical input device that returns an (x, y) screen location. The application program can ask the graphics system either to *sample* the input devices or to wait at a specified point until an *event* is generated when the user activates a device being waited on. With input values obtained from sampling or waiting for an event, the application program can handle user interactions that alter the model or the display or that change its operating mode.

1.6.4 Interaction Handling

The typical application-program schema for interaction handling is the *event-driven loop*. It is easily visualized as a finite-state machine with a central wait state and transitions to other states that are caused by user-input events. Processing a command may entail nested event loops of the same format that have their own states and input transitions. An application program may also sample input devices such as the locator by asking for their values at any time; the program then uses the returned value as input to a processing procedure that also changes the state of the application program, the image, or the database. The event-driven loop is characterized by the following pseudocode schema:

```
generate initial display, derived from application model as appropriate
while (!quit) {        /* User has not selected the "quit" option */
    enable selection of commands/objects
    /* Program pauses indefinitely in "wait state" until user acts */
    wait for user selection
    switch (selection) {
        process selection to complete command or process completed command,
            updating model and screen as needed
    }
}
```

Let's examine the application's reaction to input in more detail. The application program typically responds to user interactions in one of two modes. First, the user action may require only that the screen be updated—for example, by highlighting of a selected object

or by making available a new menu of choices. The application then needs only to update its internal state and to call the graphics package to update the screen; it does not need not to update the database. If, however, the user action calls for a change in the model—for example, by adding or deleting a component—the application must update the model and then call the graphics package to update the screen from the model. Either the entire model is retraversed to regenerate the image from scratch, or, with more sophisticated incremental-update algorithms, the screen is updated selectively. It is important to understand that no significant change can take place in the objects on the screen without a corresponding change in the model. The screen is indeed the window on the computer in that the user, in general, is manipulating not an image but the model that is literally and figuratively behind the image. Only in painting and image-enhancement applications are the model and the image identical. Therefore, it is the application's job to interpret user input. The graphics system has no responsibility for building or modifying the model, either initially or in response to user interaction; its only job is to create images from geometric descriptions and to pass along the user's input data.

The event-loop model, although fundamental to current practice in computer graphics, is limited in that the user–computer dialogue is a *sequential*, ping-pong model of alternating user actions and computer reactions. In the future, we may expect to see more of *parallel* conversations, in which simultaneous input and output using multiple communications channels—for example, both graphics and voice—take place. Formalisms, not to mention programming-language constructs, for such free-form conversations are not yet well developed; we shall not discuss them further here.

1.7 SUMMARY

Graphical interfaces have replaced textual interfaces as the standard means for user–computer interaction. Graphics has also become a key technology for communicating ideas, data, and trends in most areas of commerce, science, engineering, and education. With graphics, we can create artificial realities, each a computer-based "exploratorium" for examining objects and phenomena in a natural and intuitive way that exploits our highly developed skills in visual-pattern recognition.

Until the late eighties, the bulk of computer-graphics applications dealt with 2D objects. 3D applications were relatively rare, both because 3D software is intrinsically far more complex than is 2D software and because a great deal of computing power is required to render pseudorealistic images. Therefore, until recently, real-time user interaction with 3D models and pseudorealistic images was feasible on only very expensive high-performance workstations with dedicated, special-purpose graphics hardware. The spectacular progress of VLSI semiconductor technology that was responsible for the advent of inexpensive microprocessors and memory led in the early 1980s to the creation of 2D, bitmap-graphics–based personal computers. That same technology has made it possible, less than a decade later, to create subsystems of only a few chips that do real-time 3D animation with color-shaded images of complex objects, typically described by thousands of polygons. These subsystems can be added as 3D accelerators to workstations or even to personal computers using commodity microprocessors. It is clear that an explosive growth of 3D applications will parallel the current growth in 2D applications. Furthermore, topics

such as photorealistic rendering that were considered exotic in the 1982 edition of this book are now part of the state of the art and are available routinely in graphics software and increasingly in graphics hardware.

Much of the task of creating effective graphic communication, whether 2D or 3D, lies in modeling the objects whose images we want to produce. The graphics system acts as the intermediary between the application model and the output device. The application program is responsible for creating and updating the model based on user interaction; the graphics system does the best-understood, most routine part of the job when it creates views of objects and passes user events to the application. It is important to note that, although this separation between modeling and graphics was accepted practice at the time this book was written, our chapters on modeling (Chapters 11, 12, and 20) and animation (Chapter 21), as well as the growing literature on various types of physically based modeling, show that graphics is evolving to include a great deal more than rendering and interaction handling. Images and animations are no longer merely illustrations in science and engineering—they have become part of the content of science and engineering and are influencing how scientists and engineers conduct their daily work.

EXERCISES

1.1 List the interactive graphics programs you use on a routine basis in your "knowledge work": writing, calculating, graphing, programming, debugging, and so on. Which of these programs would work almost as well on an alphanumerics-only terminal?

1.2 The phrase "look and feel" has been applied extensively to the user interface of graphics programs. Itemize the major components—such as icons, windows, scroll bars, and menus—of the look of the graphics interface of your favorite word-processing or window-manager program. List the kinds of graphics capabilities these "widgets" require. What opportunities do you see for applying color and 3D depictions to the look? For example, how might a "cluttered office" be a more powerful spatial metaphor for organizing and accessing information than is a "messy desktop?"

1.3 In a similar vein to that of Exercise 1.2, what opportunities do you see for dynamic icons to augment or even to replace the static icons of current desktop metaphors?

1.4 Break down your favorite graphics application into its major modules, using the conceptual model of Figure 1.5 as a guide. How much of the application actually deals with graphics per se? How much deals with data-structure creation and maintainance? How much deals with calculations, such as simulation?

1.5 The terms *simulation* and *animation* are often used together and even interchangeably in computer graphics. This is natural when the behavioral (or structural) changes over time of some physical or abstract system are being visualized. Construct some examples of systems that could benefit from such visualizations. Specify what form the simulations would take and how they would be executed.

1.6 As a variation on Exercise 1.5, create a high-level design of a graphical "exploratorium" for a nontrivial topic in science, mathematics, or engineering. Discuss how the interaction sequences would work and what facilities the user should have for experimentation.

1.7 Consider an image containing a set of 10,000 1-inch unconnected vectors. Contrast the storage required for a vector display list with that for a 1-bit raster image for a 1024-by-1024 bilevel display to store this image. Assume that it takes a 8-bit "op-code" to specify "vector-draw," and four 10-bit

coordinates (i.e., 6 bytes) to store a vector in the vector display list. How do these numbers vary as a function of the number and size of the vectors, the number of bits per pixel, and the resolution of the raster display? What conclusion can you draw about the relative sizes of refresh memory required?

1.8 Without peeking at Chapter 3, construct a straightforward algorithm for scan converting a line in the first quadrant.

1.9 Aliasing is a serious problem in that it produces unpleasant or even misleading visual artifacts. Discuss situations in which these artifacts matter, and those in which they do not. Discuss various ways to minimize the effects of jaggies, and explain what the "costs" of those remedies might be.

5
Geometrical Transformations

This chapter introduces the basic 2D and 3D geometrical transformations used in computer graphics. The translation, scaling, and rotation transformations discussed here are essential to many graphics applications and will be referred to extensively in succeeding chapters.

The transformations are used directly by application programs and within many graphics subroutine packages. A city-planning application program would use translation to place symbols for buildings and trees at appropriate positions, rotation to orient the symbols, and scaling to size the symbols. In general, many applications use the geometric transformations to change the position, orientation, and size of objects (also called *symbols* or *templates*) in a drawing. In Chapter 6, 3D rotation, translation, and scaling will be used as part of the process of creating 2D renditions of 3D objects. In Chapter 7, we see how a contemporary graphics package uses transformations as part of its implementation and also makes them available to application programs.

5.1 2D TRANSFORMATIONS

We can *translate* points in the (x, y) plane to new positions by adding translation amounts to the coordinates of the points. For each point $P(x, y)$ to be moved by d_x units parallel to the x axis and by d_y units parallel to the y axis to the new point $P'(x', y')$, we can write

$$x' = x + d_x, \qquad y' = y + d_y \tag{5.1}$$

If we define the column vectors

$$P = \begin{bmatrix} x \\ y \end{bmatrix}, \; P' = \begin{bmatrix} x' \\ y' \end{bmatrix}, \; T = \begin{bmatrix} d_x \\ d_y \end{bmatrix}, \tag{5.2}$$

ATI Ex. 2003
IPR2023-00564
Page 37 of 241

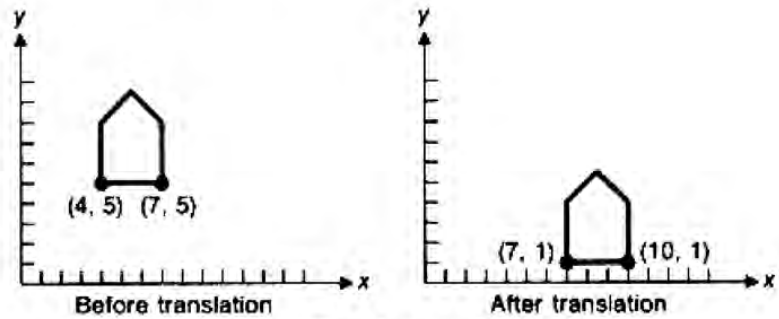


Fig. 5.1 Translation of a house.

then (5.1) can be expressed more concisely as

$$P' = P + T. \tag{5.3}$$

We could translate an object by applying Eq. (5.1) to every point of the object. Because each line in an object is made up of an infinite number of points, however, this process would take an infinitely long time. Fortunately, we can translate all the points on a line by translating only the line's endpoints and by drawing a new line between the translated endpoints; this is also true of scaling (stretching) and rotation. Figure 5.1 shows the effect of translating the outline of a house by (3, −4).

Points can be *scaled* (stretched) by s_x along the x axis and by s_y along the y axis into new points by the multiplications

$$x' = s_x \cdot x, \qquad y' = s_y \cdot y. \tag{5.4}$$

In matrix form, this is

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} \text{ or } P' = S \cdot P, \tag{5.5}$$

where S is the matrix in Eq. (5.5).

In Fig. 5.2, the house is scaled by $\frac{1}{2}$ in x and $\frac{1}{4}$ in y. Notice that the scaling is about the origin: The house is smaller *and* is closer to the origin. If the scale factors were greater than 1, the house would be both larger and further from the origin. Techniques for scaling about some point other than the origin are discussed in Section 5.2. The proportions of the house have also changed: a *differential* scaling, in which $s_x \neq s_y$, has been used. With a *uniform* scaling, in which $s_x = s_y$, the proportions are unaffected.

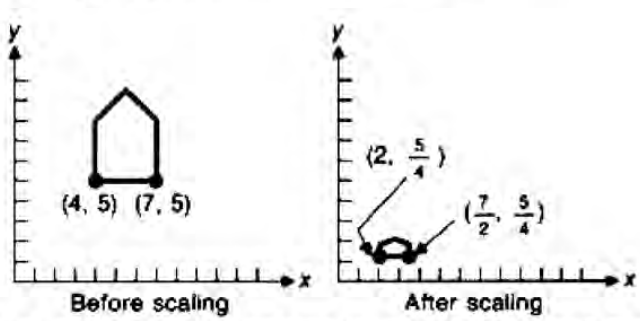


Fig. 5.2 Scaling of a house. The scaling is nonuniform, and the house changes position.

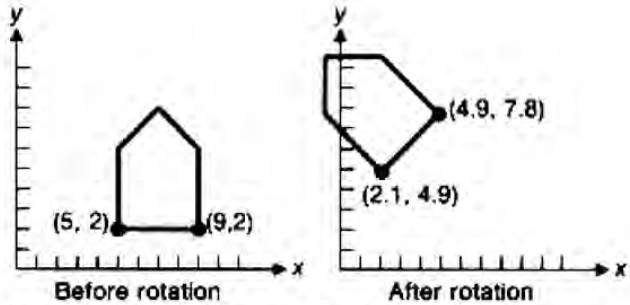


Fig. 5.3 Rotation of a house. The house also changes position.

Points can be *rotated* through an angle θ about the origin. A rotation is defined mathematically by

$$x' = x \cdot \cos\theta - y \cdot \sin\theta, \quad y' = x \cdot \sin\theta + y \cdot \cos\theta. \tag{5.6}$$

In matrix form, we have

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} \text{ or } P' = R \cdot P. \tag{5.7}$$

where R is the rotation matrix in Eq. (5.7). Figure 5.3 shows the rotation of the house by 45°. As with scaling, rotation is about the origin; rotation about an arbitrary point is discussed in Section 5.2.

Positive angles are measured *counterclockwise* from x toward y. For negative (clockwise) angles, the identities $\cos(-\theta) = \cos\theta$ and $\sin(-\theta) = -\sin\theta$ can be used to modify Eqs. (5.6) and (5.7).

Equation (5.6) is easily derived from Fig. 5.4, in which a rotation by θ transforms $P(x, y)$ into $P'(x', y')$. Because the rotation is about the origin, the distances from the origin to P and to P', labeled r in the figure, are equal. By simple trigonometry, we find that

$$x = r \cdot \cos\phi, \; y = r \cdot \sin\phi \tag{5.8}$$

and

$$\begin{aligned} x' &= r \cdot \cos(\theta + \phi) = r \cdot \cos\phi \cdot \cos\theta - r \cdot \sin\phi \cdot \sin\theta, \\ y' &= r \cdot \sin(\theta + \phi) = r \cdot \cos\phi \cdot \sin\theta + r \cdot \sin\phi \cdot \cos\theta. \end{aligned} \tag{5.9}$$

Substituting Eq. (5.8) into Eq. (5.9) yields Eq. (5.6).

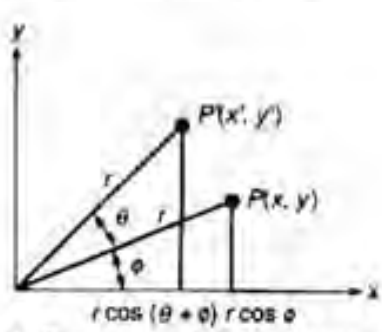


Fig. 5.4 Derivation of the rotation equation.

5.2 HOMOGENEOUS COORDINATES AND MATRIX REPRESENTATION OF 2D TRANSFORMATIONS

The matrix representations for translation, scaling, and rotation are, respectively,

$$P' = T + P, \tag{5.3}$$
$$P' = S \cdot P, \tag{5.5}$$
$$P' = R \cdot P. \tag{5.7}$$

Unfortunately, translation is treated differently (as an addition) from scaling and rotation (as multiplications). We would like to be able to treat all three transformations in a consistent way, so that they can be combined easily.

If points are expressed in *homogeneous coordinates*, all three transformations can be treated as multiplications. Homogeneous coordinates were first developed in geometry [MAXW46; MAXW51] and have been applied subsequently in graphics [ROBE65; BLIN77b; BLIN78a]. Numerous graphics subroutine packages and display processors work with homogeneous coordinates and transformations.

In homogeneous coordinates, we add a third coordinate to a point. Instead of being represented by a pair of numbers (x, y), each point is represented by a triple (x, y, W). At the same time, we say that two sets of homogeneous coordinates (x, y, W) and (x', y', W') represent the same point if and only if one is a multiple of the other. Thus, (2, 3, 6) and (4, 6, 12) are the same points represented by different coordinate triples. That is, each point has many different homogeneous coordinate representations. Also, at least one of the homogeneous coordinates must be nonzero: (0, 0, 0) is not allowed. If the W coordinate is nonzero, we can divide through by it: (x, y, W) represents the same point as $(x/W, y/W, 1)$. When W is nonzero, we normally do this division, and the numbers x/W and y/W are called the Cartesian coordinates of the homogeneous point. The points with $W = 0$ are called points at infinity, and will not appear very often in our discussions.

Triples of coordinates typically represent points in 3-space, but here we are using them to represent points in 2-space. The connection is this: If we take all the triples representing the same point—that is, all triples of the form (tx, ty, tW), with $t \neq 0$—we get a line in 3-space. Thus, each homogeneous *point* represents a *line* in 3-space. If we *homogenize* the point (divide by W), we get a point of the form $(x, y, 1)$. Thus, the homogenized points form the plane defined by the equation $W = 1$ in (x, y, W)-space. Figure 5.5 shows this

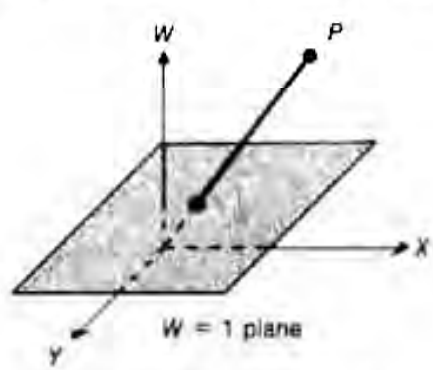


Fig. 5.5 The XYW homogeneous coordinate space, with the $W = 1$ plane and point $P(X, Y, W)$ projected onto the $W = 1$ plane.

relationship. Points at infinity are not represented on this plane.

Because points are now three-element column vectors, transformation matrices, which multiply a point vector to produce another point vector, must be 3×3. In the 3×3 matrix form for homogeneous coordinates, the translation equations Eq. (5.1) are

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & d_x \\ 0 & 1 & d_y \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}. \tag{5.10}$$

We caution the reader that some graphics textbooks, including [FOLE82], use a convention of premultiplying matrices by row vectors, rather than postmultiplying by column vectors. Matrices must be transposed to go from one convention to the other, just as the row and column vectors are transposed:

$$(P \cdot M)^T = M^T \cdot P^T.$$

Equation (5.10) can be expressed differently as

$$P' = T(d_x, d_y) \cdot P, \tag{5.11}$$

where

$$T(d_x, d_y) = \begin{bmatrix} 1 & 0 & d_x \\ 0 & 1 & d_y \\ 0 & 0 & 1 \end{bmatrix}. \tag{5.12}$$

What happens if a point P is translated by $T(d_{x1}, d_{y1})$ to P' and then translated by $T(d_{x2}, d_{y2})$ to P''? The result we expect intuitively is a net translation $T(d_{x1} + d_{x2}, d_{y1} + d_{y2})$. To confirm this intuition, we start with the givens:

$$P' = T(d_{x1}, d_{y1}) \cdot P, \tag{5.13}$$
$$P'' = T(d_{x2}, d_{y2}) \cdot P'. \tag{5.14}$$

Now, substituting Eq. (5.13) into Eq. (5.14), we obtain

$$P'' = T(d_{x2}, d_{y2}) \cdot (T(d_{x1}, d_{y1}) \cdot P) = (T(d_{x2}, d_{y2}) \cdot T(d_{x1}, d_{y1})) \cdot P. \tag{5.15}$$

The matrix product $T(d_{x2}, d_{y2}) \cdot T(d_{x1}, d_{y1})$ is

$$\begin{bmatrix} 1 & 0 & d_{x2} \\ 0 & 1 & d_{y2} \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & d_{x1} \\ 0 & 1 & d_{y1} \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & d_{x1} + d_{x2} \\ 0 & 1 & d_{y1} + d_{y2} \\ 0 & 0 & 1 \end{bmatrix}. \tag{5.16}$$

The net translation is indeed $T(d_{x1} + d_{x2}, d_{y1} + d_{y2})$. The matrix product is variously referred to as the *compounding*, *catenation*, *concatenation*, or *composition* of $T(d_{x1}, d_{y1})$ and $T(d_{x2}, d_{y2})$. Here, we shall normally use the term *composition*.

Similarly, the scaling equations Eq. (5.4) are represented in matrix form as

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}. \tag{5.17}$$

Defining

$$S(s_x, s_y) = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}, \tag{5.18}$$

we have

$$P' = S(s_x, s_y) \cdot P. \tag{5.19}$$

Just as successive translations are additive, we expect that successive scalings should be multiplicative. Given

$$P' = S(s_{x1}, s_{y1}) \cdot P, \tag{5.20}$$

$$P'' = S(s_{x2}, s_{y2}) \cdot P', \tag{5.21}$$

then, substituting Eq. (5.20) into Eq. (5.21), we get

$$P'' = S(s_{x2}, s_{y2}) \cdot (S(s_{x1}, s_{y1}) \cdot P) = (S(s_{x2}, s_{y2}) \cdot S(s_{x1}, s_{y1})) \cdot P. \tag{5.22}$$

The matrix product $S(s_{x2}, s_{y2}) \cdot S(s_{x1}, s_{y1})$ is

$$\begin{bmatrix} s_{x2} & 0 & 0 \\ 0 & s_{y2} & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} s_{x1} & 0 & 0 \\ 0 & s_{y1} & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} s_{x1} \cdot s_{x2} & 0 & 0 \\ 0 & s_{y1} \cdot s_{y2} & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{5.23}$$

Thus, the scalings are indeed multiplicative.

Finally, the rotation equations Eq. (5.6) can be represented as

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}. \tag{5.24}$$

Letting

$$R(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}, \tag{5.25}$$

we have

$$P' = R(\theta) \cdot P. \tag{5.26}$$

Showing that two successive rotations are additive is left as Exercise 5.2.

In the upper-left 2×2 submatrix of Eq. (5.25), consider each of the two rows as vectors. The vectors can be shown to have three properties:

1. Each is a unit vector
2. Each is perpendicular to the other (their dot product is zero)
3. The first and second vectors will be rotated by $R(\theta)$ to lie on the positive x and y axes, respectively (in the presence of conditions 1 and 2, this is equivalent to the submatrix having a determinant of 1).

The first two properties are also true of the columns of the 2×2 submatrix. The two directions are those into which vectors along the positive x and y axes are rotated. These properties suggest two useful ways to go about deriving a rotation matrix when we know the effect desired from the rotation. A matrix having these properties is called *special orthogonal*.

A transformation matrix of the form

$$\begin{bmatrix} r_{11} & r_{12} & t_x \\ r_{21} & r_{22} & t_y \\ 0 & 0 & 1 \end{bmatrix}, \tag{5.27}$$

where the upper 2×2 submatrix is orthogonal, preserves angles and lengths. That is, a unit square remains a unit square, and becomes neither a rhombus with unit sides, nor a square with nonunit sides. Such transformations are also called *rigid-body* transformations, because the body or object being transformed is not distorted in any way. An arbitrary sequence of rotation and translation matrices creates a matrix of this form.

What can be said about the product of an arbitrary sequence of rotation, translation, and scale matrices? They are called *affine* transformations, and have the property of preserving parallelism of lines, but not lengths and angles. Figure 5.6 shows the results of applying a $-45°$ rotation and then a nonuniform scaling to the unit cube. It is clear that neither angles nor lengths have been preserved by this sequence, but parallel lines have remained parallel. Further rotation, scale, and translation operations will not cause the parallel lines to cease being parallel. $R(\theta)$, $S(s_x, s_y)$, and $T(d_x, d_y)$ are also affine.

Another type of primitive transformation, *shear transformations*, are also affine. Two-dimensional shear transformations are of two kinds: a shear along the x axis and a shear along the y axis. Figure 5.7 shows the effect of shearing the unit cube along each axis. The operation is represented by the matrix

$$SH_x = \begin{bmatrix} 1 & a & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{5.28}$$

The term a in the shear matrix is the proportionality constant. Notice that the product $SH_x [x \quad y \quad 1]^T$ is $[x + ay \quad y \quad 1]^T$, clearly demonstrating the proportional change in x as a function of y.

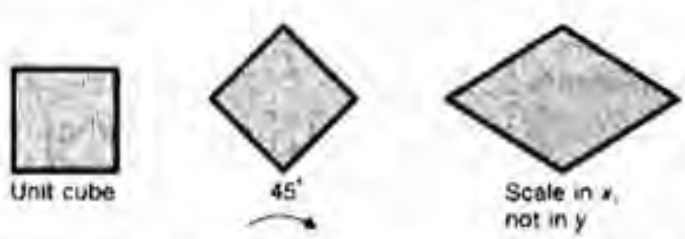


Fig. 5.6 A unit cube is rotated by $-45°$ and is nonuniformly scaled. The result is an affine transformation of the unit cube, in which parallelism of lines is maintained, but neither angles nor lengths are maintained.

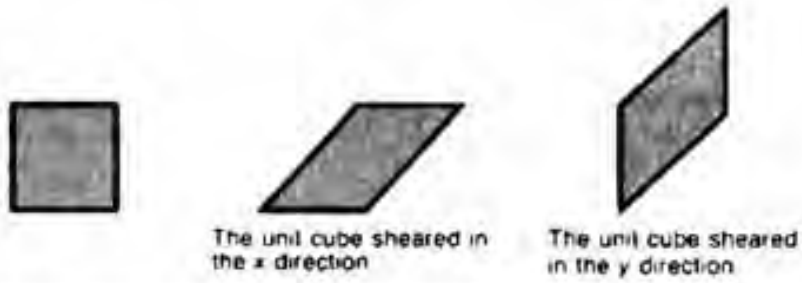


Fig. 5.7 The primitive-shear operations applied to the unit cube. In each case, the lengths of the oblique lines are now greater than 1.

Similarly, the matrix

$$SH_y = \begin{bmatrix} 1 & 0 & 0 \\ b & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{5.29}$$

shears along the y axis.

5.3 COMPOSITION OF 2D TRANSFORMATIONS

The idea of composition was introduced in the preceding section. Here, we use composition to combine the fundamental R, S, and T matrices to produce desired general results. The basic purpose of composing transformations is to gain efficiency by applying a single composed transformation to a point, rather than applying a series of transformations, one after the other.

Consider the rotation of an object about some arbitrary point P_1. Because we know how to rotate only about the origin, we convert our original (difficult) problem into three separate (easy) problems. Thus, to rotate about P_1, we need a sequence of three fundamental transformations:

1. Translate such that P_1 is at the origin
2. Rotate
3. Translate such that the point at the origin returns to P_1.

This sequence is illustrated in Fig. 5.8, in which our house is rotated about $P_1(x_1, y_1)$. The first translation is by $(-x_1, -y_1)$, whereas the later translation is by the inverse (x_1, y_1). The result is rather different from that of applying just the rotation.

The net transformation is

$$T(x_1, y_1) \cdot R(\theta) \cdot T(-x_1, -y_1) = \begin{bmatrix} 1 & 0 & x_1 \\ 0 & 1 & y_1 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & -x_1 \\ 0 & 1 & -y_1 \\ 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} \cos\theta & -\sin\theta & x_1(1-\cos\theta) + y_1\sin\theta \\ \sin\theta & \cos\theta & y_1(1-\cos\theta) - x_1\sin\theta \\ 0 & 0 & 1 \end{bmatrix}. \tag{5.30}$$

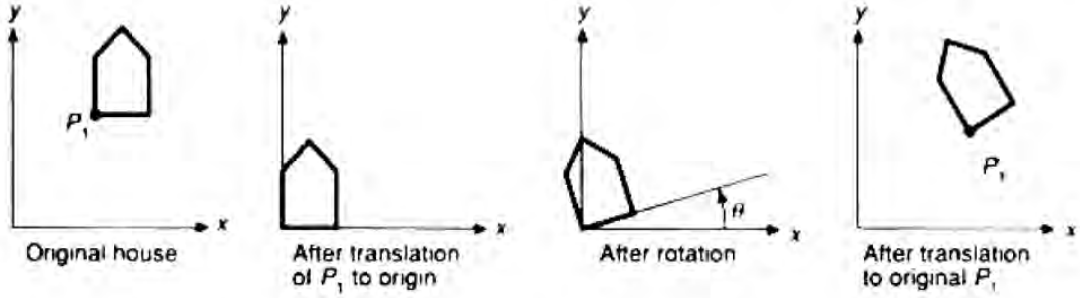


Fig. 5.8 Rotation of a house about the point P_1 by an angle θ.

A similar approach is used to scale an object about an arbitrary point P_1. First, translate such that P_1 goes to the origin, then scale, then translate back to P_1. In this case, the net transformation is

$$T(x_1, y_1) \cdot S(s_x, s_y) \cdot T(-x_1, -y_1) = \begin{bmatrix} 1 & 0 & x_1 \\ 0 & 1 & y_1 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & -x_1 \\ 0 & 1 & -y_1 \\ 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} s_x & 0 & x_1(1 - s_x) \\ 0 & s_y & y_1(1 - s_y) \\ 0 & 0 & 1 \end{bmatrix}. \tag{5.31}$$

Suppose we wish to scale, rotate, and position the house shown in Fig. 5.9 with P_1 as the center for the rotation and scaling. The sequence is to translate P_1 to the origin, to perform the scaling and rotation, and then to translate from the origin to the new position P_2 where the house is to be placed. A data structure that records this transformation might contain the scale factor(s), rotation angle, and translation amounts, and the order in which the transformations were applied, or it might simply record the composite transformation matrix:

$$T(x_2, y_2) \cdot R(\theta) \cdot S(s_x, s_y) \cdot T(-x_1, -y_1). \tag{5.32}$$

If M_1 and M_2 each represent a fundamental translation, scaling, or rotation, when is $M_1 \cdot M_2 = M_2 \cdot M_1$? That is, when do M_1 and M_2 commute? In general, of course, matrix multiplication is *not* commutative. However, it is easy to show that, in the following special cases, commutativity holds:

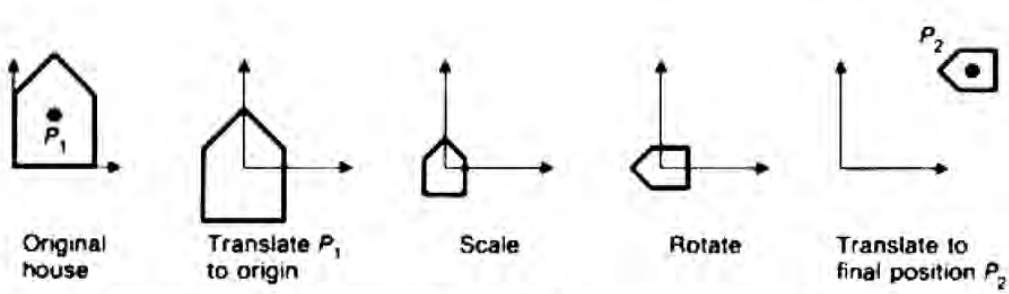


Fig. 5.9 Rotation of a house about the point P_1, and placement such that what was at P_1 is at P_2.

M_1	M_2
Translate	Translate
Scale	Scale
Rotate	Rotate
Scale (with $s_x = s_y$)	Rotate

In these cases, we need not be concerned about the *order* of matrix manipulation.

5.4 THE WINDOW-TO-VIEWPORT TRANSFORMATION

Some graphics packages allow the programmer to specify output primitive coordinates in a floating-point *world-coordinate* system, using whatever units are meaningful to the application program: angstroms, microns, meters, miles, light-years, and so on. The term *world* is used because the application program is representing a world that is being interactively created or displayed to the user.

Given that output primitives are specified in world coordinates, the graphics subroutine package must be told how to map world coordinates onto screen coordinates (we use the specific term *screen coordinates* to relate this discussion specifically to SRGP, but that hardcopy output devices might be used, in which case the term *device coordinates* would be more appropriate). We could do this mapping by having the application programmer provide the graphics package with a transformation matrix to effect the mapping. Another way is to have the application programmer specify a rectangular region in world coordinates, called the *world-coordinate window*, and a corresponding rectangular region in screen coordinates, called the *viewport*, into which the world-coordinate window is to be mapped. The transformation that maps the window into the viewport is applied to all of the output primitives in world coordinates, thus mapping them into screen coordinates. Figure 5.10 shows this concept. As seen in this figure, if the window and viewport do not have the same height-to-width ratio, a *non*uniform scaling occurs. If the application program changes the window or viewport, then new output primitives drawn onto the screen will be affected by the change. Existing output primitives are not affected by such a change.

The modifier *world-coordinate* is used with *window* to emphasize that we are not discussing a *window-manager window*, which is a different and more recent concept, and

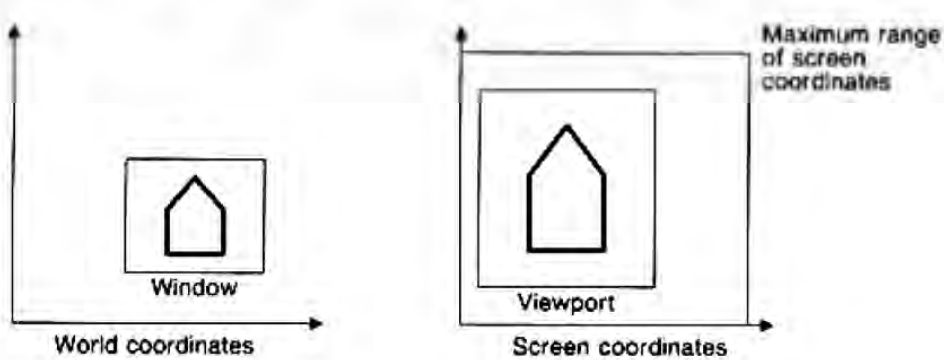


Fig. 5.10 The window in world coordinates and the viewport in screen coordinates determine the mapping that is applied to all the output primitives in world coordinates.

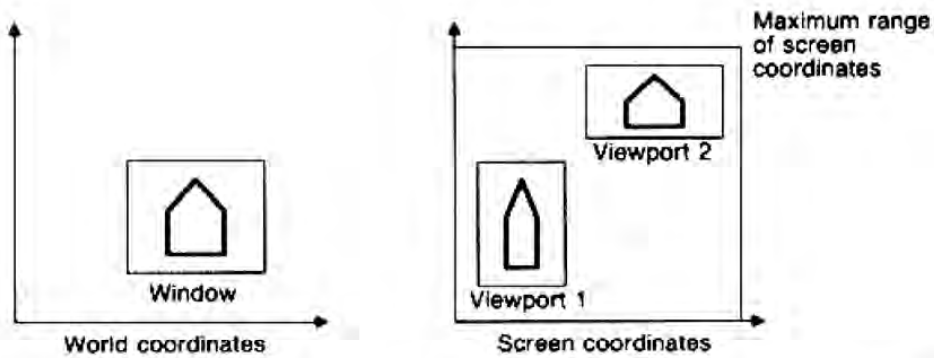


Fig. 5.11 The effect of drawing output primitives with two viewports. Output primitives specifying the house were first drawn with viewport 1, the viewport was changed to viewport 2, and then the application program again called the graphics package to draw the output primitives.

which unfortunately has the same name. Whenever there is no ambiguity as to which type of window is meant, we will drop the modifier.

If SRGP were to provide world-coordinate output primitives, the viewport would be on the current canvas, which defaults to canvas 0, the screen. The application program would be able to change the window or the viewport at any time, in which case subsequently specified output primitives would be subjected to a new transformation. If the change included a different viewport, then the new output primitves would be located on the canvas in positions different from those of the old ones, as shown in Fig. 5.11.

A window manager might map SRGP's canvas 0 into less than a full-screen window, in which case not all of the canvas or even of the viewport would necessarily be visible. In Chapter 10, we further discuss the relationships among world-coordinate windows, viewports, and window-manager windows.

Given a window and viewport, what is the transformation matrix that maps the window from world coordinates into the viewport in screen coordinates? This matrix can be developed as a three-step transformation composition, as suggested in Fig. 5.12. The

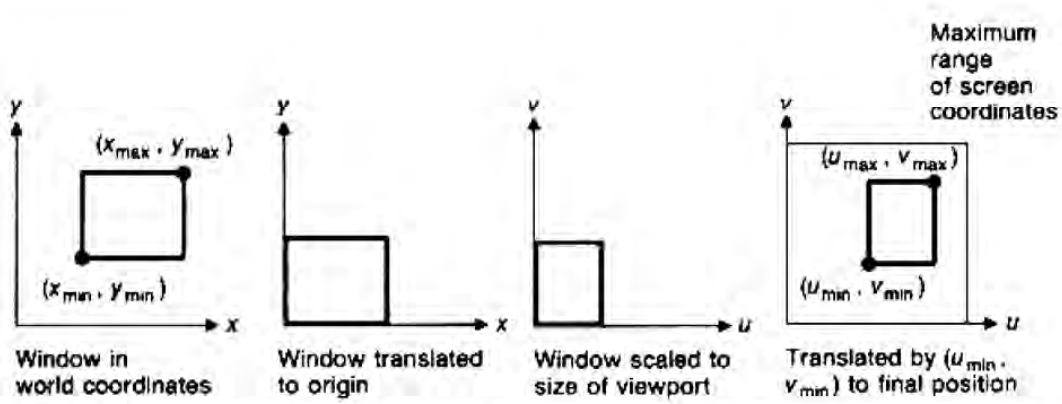


Fig. 5.12 The steps in transforming a world-coordinate window into a viewport.

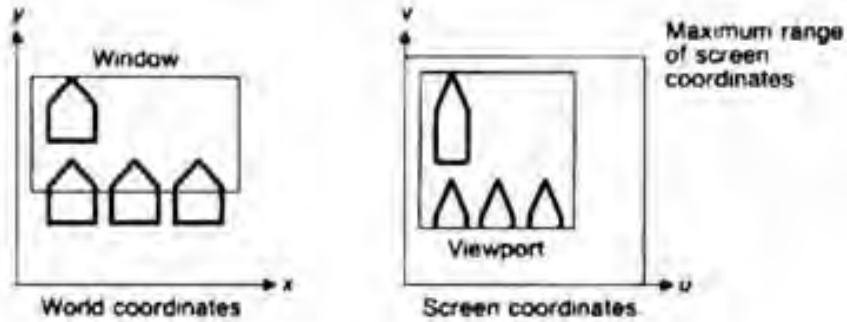


Fig. 5.13 Output primitives in world coordinates are clipped against the window. Those that remain are displayed in the viewport.

window, specified by its lower-left and upper-right corners, is first translated to the origin of world coordinates. Next, the size of the window is scaled to be equal to the size of the viewport. Finally, a translation is used to position the viewport. The overall matrix M_{wv} is:

$$M_{wv} = T(u_{min}, v_{min}) \cdot S\left(\frac{u_{max} - u_{min}}{x_{max} - x_{min}}, \frac{v_{max} - v_{min}}{y_{max} - y_{min}}\right) \cdot T(-x_{min}, -y_{min})$$

$$= \begin{bmatrix} 1 & 0 & u_{min} \\ 0 & 1 & v_{min} \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \frac{u_{max} - u_{min}}{x_{max} - x_{min}} & 0 & 0 \\ 0 & \frac{v_{max} - v_{min}}{y_{max} - y_{min}} & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & -x_{min} \\ 0 & 1 & -y_{min} \\ 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} \frac{u_{max} - u_{min}}{x_{max} - x_{min}} & 0 & -x_{min} \cdot \frac{u_{max} - u_{min}}{x_{max} - x_{min}} + u_{min} \\ 0 & \frac{v_{max} - v_{min}}{y_{max} - y_{min}} & -y_{min} \cdot \frac{v_{max} - v_{min}}{y_{max} - y_{min}} + v_{min} \\ 0 & 0 & 1 \end{bmatrix}. \tag{5.33}$$

Multiplying $P = M_{wv}\,[x \quad y \quad 1]^T$ gives the expected result:

$$P = \left[(x - x_{min}) \cdot \frac{u_{max} - u_{min}}{x_{max} - x_{min}} + u_{min} \quad (y - y_{min}) \cdot \frac{v_{max} - v_{min}}{y_{max} - y_{min}} + v_{min} \quad 1\right]. \tag{5.34}$$

Many graphics packages combine the window–viewport transformation with clipping of output primitives against the window. The concept of clipping was introduced in Chapter 3; Fig. 5.13 illustrates clipping in the context of windows and viewports.

5.5 EFFICIENCY

The most general composition of R, S, and T operations produces a matrix of the form

$$M = \begin{bmatrix} r_{11} & r_{12} & t_x \\ r_{21} & r_{22} & t_y \\ 0 & 0 & 1 \end{bmatrix}. \tag{5.35}$$

The upper 2×2 submatrix is a composite rotation and scale matrix, whereas t_x and t_y are composite translations. Calculating $M \cdot P$ as a vector multiplied by a 3×3 matrix takes nine multiplies and six adds. The fixed structure of the last row of Eq. (5.35), however, simplifies the actual operations to

$$x' = x \cdot r_{11} + y \cdot r_{12} + t_x \qquad (5.36)$$
$$y' = x \cdot r_{21} + y \cdot r_{22} + t_y,$$

reducing the process to four multiplies and four adds—a significant speedup, especially since the operation can be applied to hundreds or even thousands of points per picture. Thus, although 3×3 matrices are convenient and useful for composing 2D transformations, we can use the final matrix most efficiently in a program by exploiting its special structure. Some hardware matrix multipliers have parallel adders and multipliers, thereby diminishing or removing this concern.

Another area where efficiency is important is creating successive views of an object, such as a molecule or airplane, rotated a few degrees between each successive view. If each view can be created and displayed quickly enough (30 to 100 milliseconds each), then the object will appear to be rotating dynamically. To achieve this speed, we must transform each individual point and line of the object as quickly as possible. The rotation equations (Eq. (5.6)) require four multiplies and two adds. We can decrease the operation count by recognizing that, because θ is small (just a few degrees), $\cos\theta$ is very close to 1. In this approximation, Eq. (5.6) becomes

$$x' = x - y \sin\theta, \qquad y' = x \sin\theta + y, \qquad (5.37)$$

which requires just two multiplies and two adds. The savings of two multiplies can be significant on computers lacking hardware multipliers.

Equation (5.37), however, is only an approximation to the correct values of x' and y': a small error is built in. Each time the formulae are applied to the new values of x and y, the error gets a bit larger. If we repeat the formulae indefinitely, the error will overwhelm the correct values, and the rotating image will begin to look like a collection of randomly drawn lines.

A better approximation is to use x' instead of x in the second equation:

$$x' = x - y \sin\theta,$$
$$y' = x' \sin\theta + y = (x - y \sin\theta)\sin\theta + y = x \sin\theta + y(1 - \sin^2\theta) \qquad (5.38)$$

This is a better approximation than is Eq. (5.37) because the determinant of the corresponding 2×2 matrix is 1, which means that the areas transformed by Eq. (5.38) are unchanged. Note that cumulative errors can also arise when using the correct rotation equations repeatedly (see Exercise 5.19).

5.6 MATRIX REPRESENTATION OF 3D TRANSFORMATIONS

Just as 2D transformations can be represented by 3×3 matrices using homogeneous coordinates, so 3D transformations can be represented by 4×4 matrices, providing we use homogeneous coordinate representations of points in 3-space as well. Thus, instead of representing a point as (x, y, z), we represent it as (x, y, z, W), where two of these

quadruples represent the same point if one is a nonzero multiple of the other; the quadruple $(0, 0, 0, 0)$ is not allowed. As in 2D, a standard representation of a point (x, y, z, W) with $W \neq 0$ is given by $(x/W, y/W, z/W, 1)$. Transforming the point to this form is called *homogenizing*, as before. Also, points whose W coordinate is zero are called points at infinity. There is a geometrical interpretation as well. Each point in 3-space is being represented by a line through the origin in 4-space, and the homogenized representations of these points form a 3D subspace of 4-space which is defined by the single equation $W = 1$.

The 3D coordinate system used in this text is *right-handed*, as shown in Fig. 5.14. By convention, positive rotations in a right-handed system are such that, when looking from a positive axis toward the origin, a 90° *counterclockwise* rotation will transform one positive axis into the other. This table follows from this convention:

If axis of rotation is	Direction of positive rotation is
x	y to z
y	z to x
z	x to y

These positive directions are also depicted in Fig. 5.14. The reader is warned that not all graphics texts follow this convention.

We use a right-handed system here because it is the standard mathematical convention, even though it is convenient in 3D graphics to think of a left-handed system superimposed on the face of a display (see Fig. 5.15), since a left-handed system gives the natural interpretation that larger z values are further from the viewer. Notice that, in a left-handed system, positive rotations are *clockwise* when looking from a positive axis toward the origin. This definition of positive rotations allows the same rotation matrices given in this section to be used for either right- or left-hand coordinate systems. Conversion from right to left and left to right is discussed in Section 5.8.

Translation in 3D is a simple extension from that in 2D:

$$T(d_x, d_y, d_z) = \begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{5.39}$$

That is, $T(d_x, d_y, d_z) \cdot [x \quad y \quad z \quad 1]^T = [x + d_x \quad y + d_y \quad z + d_z \quad 1]^T$.

Fig. 5.14 The right-handed coordinate system

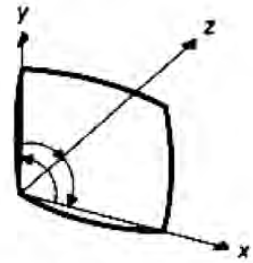


Fig. 5.15 The left-handed coordinate system, with a superimposed display screen.

Scaling is similarly extended:

$$S(s_x, s_y, s_z) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{5.40}$$

Checking, we see that $S(s_x, s_y, s_z) \cdot [x \quad y \quad z \quad 1]^T = [s_x \cdot x \quad s_y \cdot y \quad s_z \cdot z \quad 1]^T$.

The 2D rotation of Eq. (5.26) is just a 3D rotation about the z axis, which is

$$R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{5.41}$$

This is easily verified: A 90° rotation of $[1\ 0\ 0\ 1]^T$, which is the unit vector along the x axis, should produce the unit vector $[0\ 1\ 0\ 1]^T$ along the y axis. Evaluating the product

$$\begin{bmatrix} 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix} \tag{5.42}$$

gives the predicted result of $[0\ 1\ 0\ 1]^T$.

The x-axis rotation matrix is

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{5.43}$$

The y-axis rotation matrix is

$$R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{5.44}$$

The columns (and the rows) of the upper-left 3×3 submatrix of $R_z(\theta)$, $R_x(\theta)$, and $R_y(\theta)$ are mutually perpendicular unit vectors and the submatrix has a determinant of 1, which means the three matrices are special orthogonal, as discussed in Section 5.2. Also, the upper-left 3×3 submatrix formed by an arbitrary sequence of rotations is special orthogonal. Recall that orthogonal transformations preserve distances and angles.

All these transformation matrices have inverses. The inverse for T is obtained by negating d_x, d_y, and d_z; for S, by replacing s_x, s_y, and s_z by their reciprocals; that for each of the three rotation matrices is obtained by negating the angle of rotation.

The inverse of any orthogonal matrix B is just B's transpose: $B^{-1} = B^{T}$. In fact, taking the transpose does not need to involve even exchanging elements in the array that stores the matrix—it is necessary only to exchange row and column indexes when accessing the array. Notice that this method of finding an inverse is consistent with the result of negating θ to find the inverse of R_x, R_y, and R_z.

Any number of rotation, scaling, and translation matrices can be multiplied together. The result always has the form

$$M = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{5.45}$$

As in the 2D case, the 3×3 upper-left submatrix R gives the aggregate rotation and scaling, whereas T gives the subsequent aggregate translation. Some computational efficiency is achieved by performing the transformation explicitly as

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = R \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix} + T, \tag{5.46}$$

where R and T are submatrices from Eq. (5.45).

Corresponding to the two-dimensional shear matrices in Section 5.2 are three 3D shear matrices. The (x, y) shear is

$$SH_{xy}(sh_x, sh_y) = \begin{bmatrix} 1 & 0 & sh_x & 0 \\ 0 & 1 & sh_y & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{5.47}$$

Applying SH_{xy} to the point $[x \quad y \quad z \quad 1]^{T}$, we have $[x + sh_x \cdot z \quad y + sh_y \cdot z \quad z \quad 1]^{T}$. Shears along the x and y axes have a similar form.

So far, we have focused on transforming individual points. We transform lines, these being defined by two points, by transforming the endpoints. Planes, if they are defined by three points, may be handled the same way, but usually they are defined by a plane equation, and the coefficients of this plane equation must be transformed differently. We may also need to transform the plane normal. Let a plane be represented as the column vector of plane-equation coefficients $N = [A \quad B \quad C \quad D]^{T}$. Then a plane is defined by all points P such that $N \cdot P = 0$, where $\cdot$ is the vector dot product and $P = [x \quad y \quad z \quad 1]^{T}$. This dot product gives rise to the familiar plane equation $A\,x + B\,y + C\,z + D = 0$, which

can also be expressed as the product of the row vector of plane-equation coefficients times the column vector P: $N^T \cdot P = 0$. Now suppose we transform all points P on the plane by some matrix M. To maintain $N^T \cdot P = 0$ for the transformed points, we would like to transform N by some (to be determined) matrix Q, giving rise to the equation $(Q \cdot N)^T \cdot M \cdot P = 0$. This expression can in turn be rewritten as $N^T \cdot Q^T \cdot M \cdot P = 0$, using the identity $(Q \cdot N)^T = N^T \cdot Q^T$. The equation will hold if $Q^T \cdot M$ is a multiple of the identity matrix. If the multiplier is 1, this leads to $Q^T = M^{-1}$, or $Q = (M^{-1})^T$. This means that the column vector N' of coefficients for a plane transformed by M is given by

$$N' = (M^{-1})^T \cdot N.$$

The matrix $(M^{-1})^T$ need not in general exist, because the determinant of M might be zero. This would happen if M includes a projection (we might want to investigate the effect of a perspective projection on a plane). It is possible to use, instead of $(M^{-1})^T$, the matrix of cofactors of M used in finding the inverse of M using Cramer's rule. See the Appendix for more details.

If just the normal of the plane is to be transformed (for example, to perform the shading calculations discussed in Chapter 16) and if M consists of only the composition of translation, rotation, and uniform scaling matrices, then the mathematics is even simpler. The N' of Eq. (5.48) can be simplified to $[A' \; B' \; C' \; 0]^T$. (With a zero W component, a homogeneous point represents a point at infinity, which can be thought of as a direction.)

5.7 COMPOSITION OF 3D TRANSFORMATIONS

In this section, we discuss how to compose 3D transformation matrices, using an example that will be useful in Section 6.4. The objective is to transform the directed line segments P_1P_2 and P_1P_3 in Fig. 5.16 from their starting position in part (a) to their ending position in part (b). Thus, point P_1 is to be translated to the origin, P_1P_2 is to lie on the positive z axis, and P_1P_3 is to lie in the positive y axis half of the (y, z) plane. The lengths of the lines are to be unaffected by the transformation.

Two ways to achieve the desired transformation are presented. The first approach is to compose the primitive transformations T, R_x, R_y, and R_z. This approach, although somewhat tedious, is easy to illustrate, and understanding it will help us to build an understanding of transformations. The second approach, using the properties of orthogonal matrices described in the previous section, is explained more briefly but is more abstract.

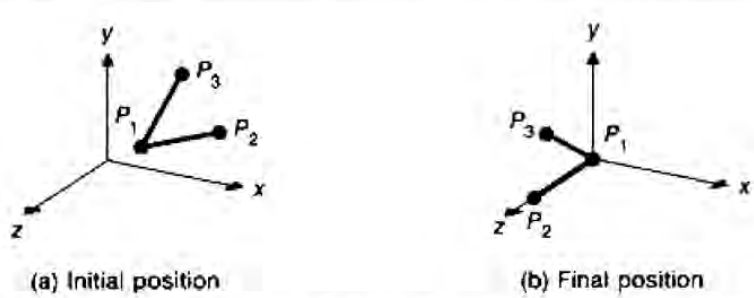


Fig. 5.16 Transforming P_1, P_2, and P_3 from their initial (a) to their final (b) position.

To work with the primitive transformations, we again break a difficult problem into simpler subproblems. In this case, the desired transformation can be done in four steps:

1. Translate P_1 to the origin
2. Rotate about the y axis such that P_1P_2 lies in the (y, z) plane
3. Rotate about the x axis such that P_1P_2 lies on the z axis
4. Rotate about the z axis such that P_1P_3 lies in the (y, z) plane.

Step 1: Translate P_1 to the origin. The translation is

$$T(-x_1, -y_1, -z_1) = \begin{bmatrix} 1 & 0 & 0 & -x_1 \\ 0 & 1 & 0 & -y_1 \\ 0 & 0 & 1 & -z_1 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{5.49}$$

Applying T to P_1, P_2, and P_3 gives

$$P'_1 = T(-x_1, -y_1, -z_1) \cdot P_1 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}, \tag{5.50}$$

$$P'_2 = T(-x_1, -y_1, -z_1) \cdot P_2 = \begin{bmatrix} x_2 - x_1 \\ y_2 - y_1 \\ z_2 - z_1 \\ 1 \end{bmatrix}, \tag{5.51}$$

$$P'_3 = T(-x_1, -y_1, -z_1) \cdot P_3 = \begin{bmatrix} x_3 - x_1 \\ y_3 - y_1 \\ z_3 - z_1 \\ 1 \end{bmatrix}. \tag{5.52}$$

Step 2: Rotate about the y axis. Figure 5.17 shows P_1P_2 after step 1, along with the projection of P_1P_2 onto the (x, z) plane. The angle of rotation is $-(90 - \theta) = \theta - 90$. Then

$$\cos(\theta - 90) = \sin\theta = \frac{z_2'}{D_1} = \frac{z_2 - z_1}{D_1},$$
$$\sin(\theta - 90) = -\cos\theta = -\frac{x_2'}{D_1} = -\frac{x_2 - x_1}{D_1}, \tag{5.53}$$

where

$$D_1 = \sqrt{(z_2')^2 + (x_2')^2} = \sqrt{(z_2 - z_1)^2 + (x_2 - x_1)^2}. \tag{5.54}$$

When these values are substituted into Eq. (5.44), we get

$$P_2'' = R_y(\theta - 90) \cdot P_2' = [0 \quad y_2 - y_1 \quad D_1 \quad 1]^T. \tag{5.55}$$

As expected, the x component of P_2'' is zero, and the z component is the length D_1.

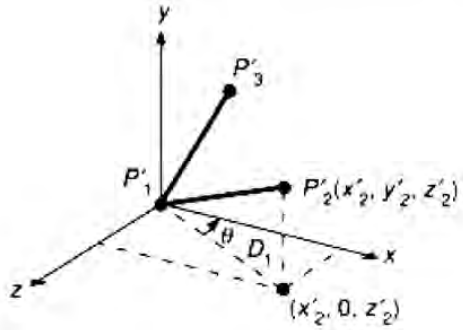


Fig. 5.17 Rotation about the y axis: The projection of $P_1'P_2'$, which has length D_1, is rotated into the z axis. The angle θ shows the positive direction of rotation about the y axis: The actual angle used is $-(90 - \theta)$.

Step 3: Rotate about the x axis. Figure 5.18 shows P_1P_2 after step 2. The angle of rotation is ϕ, for which

$$\cos\phi = \frac{z_2''}{D_2}, \ \sin\phi = \frac{y_2''}{D_2}, \tag{5.56}$$

where $D_2 = |P_1''P_2''|$, the length of the line $P_1''P_2''$. But the length of line $P_1''P_2''$ is the same as the length of line P_1P_2, because rotation and translation transformations preserve length, so

$$D_2 = |P_1''P_2''| = |P_1P_2| = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}. \tag{5.57}$$

The result of the rotation in step 3 is

$$\begin{aligned} P_2''' &= R_x(\phi) \cdot P_2'' = R_x(\phi) \cdot R_y(\theta - 90) \cdot P_2' \\ &= R_x(\phi) \cdot R_y(\theta - 90) \cdot T \cdot P_2 = [0 \quad 0 \quad |P_1P_2| \quad 1]^T. \end{aligned} \tag{5.58}$$

That is, P_1P_2 now coincides with the positive z axis.

Step 4: Rotate about the z axis. Figure 5.19 shows P_1P_2 and P_1P_3 after step 3, with P_2''' on the z axis and P_3''' at the position

$$P_3''' = [x_3''' \quad y_3''' \quad z_3''' \quad 1]^T = R_x(\phi) \cdot R_y(\theta - 90) \cdot T(-x_1, -y_1, -z_1) \cdot P_3. \tag{5.59}$$

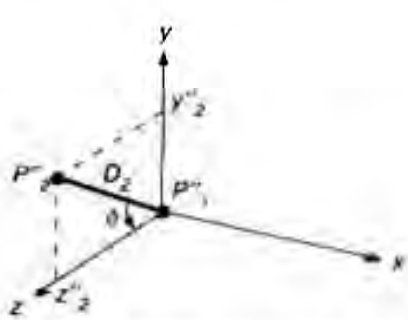


Fig. 5.18 Rotation about the x axis: $P_1''P_2''$ is rotated into the z axis by the positive angle ϕ. D_2 is the length of the line segment. The line segment $P_1''P_3''$ is not shown, because it is not used to determine the angles of rotation. Both lines are rotated by $R_x(\phi)$.

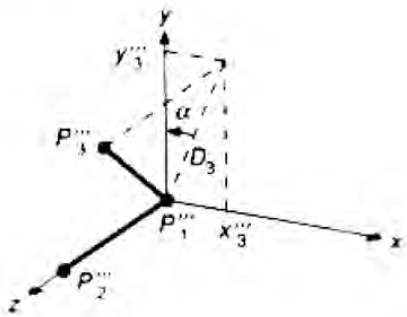


Fig. 5.19 Rotation about the z axis: The projection of $P_1'''P_3'''$, whose length is D_3, is rotated by the positive angle α into the y axis, bringing the line itself into the (y, z) plane. D_3 is the length of the projection.

The rotation is through the positive angle α, with

$$\cos\alpha = y_3'''/D_3, \quad \sin\alpha = x_3'''/D_3, \quad D_3 = \sqrt{x_3'''^2 + y_3'''^2}. \tag{5.60}$$

Step 4 achieves the result shown in Fig. 5.16(b).

The composite matrix

$$R_z(\alpha) \cdot R_x(\phi) \cdot R_y(\theta - 90) \cdot T(-x_1, -y_1, -z_1) = R \cdot T \tag{5.61}$$

is the required transformation, with $R = R_z(\alpha) \cdot R_x(\phi) \cdot R_y(\theta - 90)$. We leave it to the reader to apply this transformation to P_1, P_2, and P_3 to verify that P_1 is transformed to the origin, P_2 is transformed to the positive z axis, and P_3 is transformed to the positive y half of the (y, z) plane.

The second way to obtain the matrix R is to use the properties of orthogonal matrices discussed in Section 5.2. Recall that the unit row vectors of R rotate into the principal axes. Replacing the second subscripts of Eq. (5.45) with x, y, and z for notational convenience

$$R = \begin{bmatrix} r_{1x} & r_{2x} & r_{3x} \\ r_{1y} & r_{2y} & r_{3y} \\ r_{1z} & r_{2z} & r_{3z} \end{bmatrix}. \tag{5.62}$$

Because R_z is the unit vector along P_1P_2 that will rotate into the positive z axis,

$$R_z = [r_{1z} \quad r_{2z} \quad r_{3z}]^T = \frac{P_1P_2}{|P_1P_2|}. \tag{5.63}$$

In addition, the R_x unit vector is perpendicular to the plane of P_1, P_2, and P_3 and will rotate into the positive x axis, so that R_x must be the normalized cross-product of two vectors in the plane:

$$R_x = [r_{1x} \quad r_{2x} \quad r_{3x}]^T = \frac{P_1P_3 \times P_1P_2}{|P_1P_3 \times P_1P_2|}. \tag{5.64}$$

Finally,

$$R_y = [r_{1y} \quad r_{2y} \quad r_{3y}]^T = R_z \times R_x \tag{5.65}$$

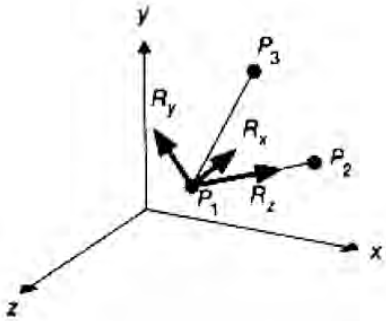


Fig. 5.20 The unit vectors R_x, R_y, and R_z, which are transformed into the principal axes.

will rotate into the positive y axis. The composite matrix is given by

$$\begin{bmatrix} r_{1_x} & r_{2_x} & r_{3_x} & 0 \\ r_{1_y} & r_{2_y} & r_{3_y} & 0 \\ r_{1_z} & r_{2_z} & r_{3_z} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot T(-x_1, -y_1, -z_1) = R \cdot T, \tag{5.66}$$

where R and T are as in Eq. (5.61). Figure 5.20 shows the individual vectors R_x, R_y, and R_z.

Let's consider another example. Figure 5.21 shows an airplane defined in the x_p, y_p, z_p coordinate system and centered at the origin. We want to transform the airplane so that it heads in the direction given by the vector *DOF* (direction of flight), is centered at P_1 and is not banked, as shown in Fig. 5.22. The transformation to do this consists of a rotation to head the airplane in the proper direction, followed by a translation from the origin to P_1. To find the rotation matrix, we just determine in what direction each of the x_p, y_p, and z_p axes is heading in Fig. 5.22, make sure the directions are normalized, and use them as column vectors in a rotation matrix.

The z_p axis must be transformed to the *DOF* direction, and the x_p axis must be transformed into a horizontal vector perpendicular to *DOF*—that is, in the direction of $y \times DOF$, the cross-product of y and *DOF*. The y_p direction is given by $z_p \times x_p = DOF \times (y \times$

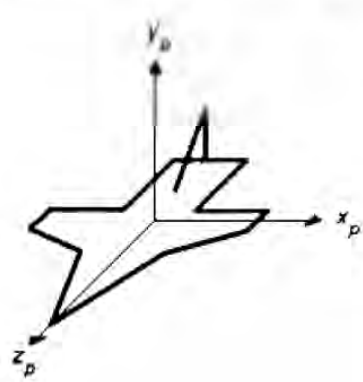


Fig. 5.21 An airplane in the (x_p, y_p, z_p) coordinate system.

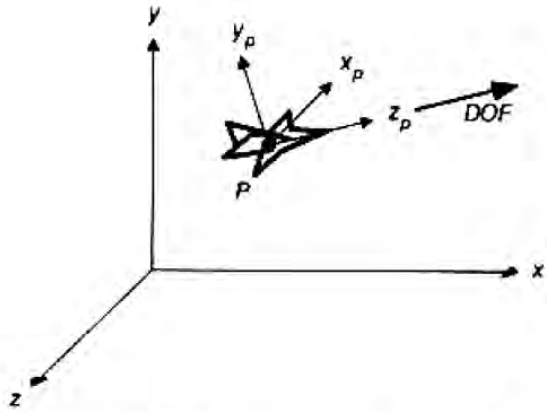


Fig. 5.22 The airplane of Figure 5.21 positioned at point P, and headed in direction DOF.

DOF), the cross-product of z_p and x_p; hence, the three columns of the rotation matrix are the normalized vectors $|y \times DOF|$, $|DOF \times (y \times DOF)|$, and $|DOF|$:

$$R = \begin{bmatrix} |y \times DOF| & |DOF \times (y \times DOF)| & |DOF| & \begin{matrix} 0 \\ 0 \\ 0 \end{matrix} \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{5.67}$$

The situation if DOF is in the direction of the y axis is degenerate, because there is an infinite set of possible vectors for the horizontal vector. This degeneracy is reflected in the algebra, because the cross-products $y \times DOF$ and $DOF \times (y \times DOF)$ are zero. In this special case, R is not a rotation matrix.

5.8 TRANSFORMATIONS AS A CHANGE IN COORDINATE SYSTEM

We have been discussing transforming a set of points belonging to an object into another set of points, when both sets are in the same coordinate system. With this approach, the coordinate system stays unaltered and the object is transformed with respect to the origin of the coordinate system. An alternative but equivalent way of thinking about a transformation is as a change of coordinate systems. This view is useful when multiple objects, each defined in its own local coordinate system, are combined, and we wish to express these objects' coordinates in a single, global coordinate system. This will be the case in Chapter 7.

Let us define $M_{i \leftarrow j}$ as the transformation that converts the representation of a point in coordinate system j into its representation in coordinate system i.

We define $P^{(i)}$ as the representation of a point in coordinate system i, $P^{(j)}$ as the representation of the point in system j, and $P^{(k)}$ as the representation of the point in coordinate system k; then,

$$P^{(i)} = M_{i \leftarrow j} \cdot P^{(j)} \text{ and } P^{(j)} = M_{j \leftarrow k} \cdot P^{(k)}. \tag{5.68}$$

Substituting, we obtain

$$P^{(i)} = M_{i\leftarrow j} \cdot P^{(j)} = M_{i\leftarrow j} \cdot M_{j\leftarrow k} \cdot P^{(k)} = M_{i\leftarrow k} \cdot P^{(k)}, \tag{5.69}$$

so

$$M_{i\leftarrow k} = M_{i\leftarrow j} \cdot M_{j\leftarrow k}. \tag{5.70}$$

Figure 5.23 shows four different coordinate systems. We see by inspection that the transformation from coordinate system 2 to 1 is $M_{1\leftarrow 2} = T(4, 2)$ (finding this transformation by inspection is not always simple—see the Appendix). Similarly, $M_{2\leftarrow 3} = T(2, 3) \cdot S(0.5, 0.5)$ and $M_{3\leftarrow 4} = T(6.7, 1.8) \cdot R(-45°)$. Then $M_{1\leftarrow 3} = M_{1\leftarrow 2} \cdot M_{2\leftarrow 3} = T(4, 2) \cdot T(2, 3) \cdot S(0.5, 0.5)$. The figure also shows a point that is $P^{(1)} = (10, 8)$, $P^{(2)} = (6, 6)$, $P^{(3)} = (8, 6)$, and $P^{(4)} = (4, 2)$ in coordinate systems 1 through 4, respectively. It is easy to verify that $P^{(i)} = M_{i\leftarrow j} \cdot P^{(j)}$ for $1 \le i, j \le 4$.

We also notice that $M_{i\leftarrow j} = M_{j\leftarrow i}^{-1}$. Thus, $M_{2\leftarrow 1} = M_{1\leftarrow 2}^{-1} = T(-4, -2)$. Because $M_{1\leftarrow 3} = M_{1\leftarrow 2} \cdot M_{2\leftarrow 3}$, $M_{1\leftarrow 3}^{-1} = M_{2\leftarrow 3}^{-1} \cdot M_{1\leftarrow 2}^{-1} = M_{3\leftarrow 2} \cdot M_{2\leftarrow 1}$.

In Section 5.6, we discussed left- and right-handed coordinate systems. The matrix that converts from points represented in one to points represented in the other is its own inverse, and is

$$M_{R\leftarrow L} = M_{L\leftarrow R} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{5.71}$$

The approach used in previous sections—defining all objects in the world-coordinate system, then transforming them to the desired place—implies the somewhat unrealistic notion that all objects are initially defined on top of one another in the same world-coordinate system. It is more natural to think of each object as being defined in its own coordinate system and then being scaled, rotated, and translated by redefinition of its coordinates in the new world-coordinate system. In this second point of view, one thinks naturally of separate pieces of paper, each with an object on it, being shrunk or stretched, rotated or placed on the world-coordinate plane. One can also, of course, imagine that the

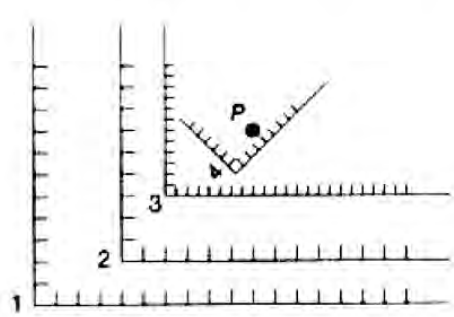


Fig. 5.23 The point P and coordinate systems 1, 2, 3, and 4

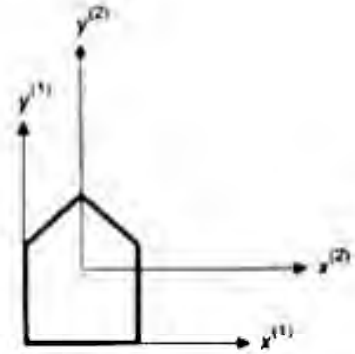


Fig. 5.24 The house and two coordinate systems. Coordinates of points on the house can be represented in either coordinate system.

plane is being shrunk or stretched, tilted, or slid relative to each piece of paper. Mathematically, all these views are identical.

Consider the simple case of translating the set of points that define the house shown in Fig. 5.8 to the origin. This transformation is $T(-x_1, -y_1)$. Labeling the two coordinate systems as in Fig. 5.24, we see that the transformation that maps coordinate system 1 into 2—that is, $M_{2\leftarrow 1}$—is $T(x_1, y_1)$, which is just $T(-x_1, -y_1)^{-1}$. Indeed, the general rule is that the transformation that transforms a set of points in a single coordinate system is just the inverse of the corresponding transformation to change the coordinate system in which a point is represented. This relation can be seen in Fig. 5.25, which is derived directly from Fig. 5.9. The transformation for the points represented in a single coordinate system is just

$$T(x_2, y_2) \cdot R(\theta) \cdot S(s_x, s_y) \cdot T(-x_1, -y_1). \tag{5.32}$$

In Fig. 5.25, the coordinate-system transformation is just

$$\begin{aligned} M_{5\leftarrow 1} &= M_{5\leftarrow 4}\, M_{4\leftarrow 3}\, M_{3\leftarrow 2}\, M_{2\leftarrow 1} \\ &= (T(x_2, y_2) \cdot R(\theta) \cdot S(s_x, s_y) \cdot T(-x_1, -y_1))^{-1} \\ &= T(x_1, y_1) \cdot S(s_x^{-1}, s_y^{-1}) \cdot R(-\theta) \cdot T(-x_2, -y_2), \end{aligned} \tag{5.72}$$

so that

$$P^{(5)} = M_{5\leftarrow 1}P^{(1)} = T(x_1, y_1) \cdot S(s_x^{-1}, s_y^{-1}) \cdot R(-\theta) \cdot T(-x_2, -y_2) \cdot P^{(1)}. \tag{5.73}$$

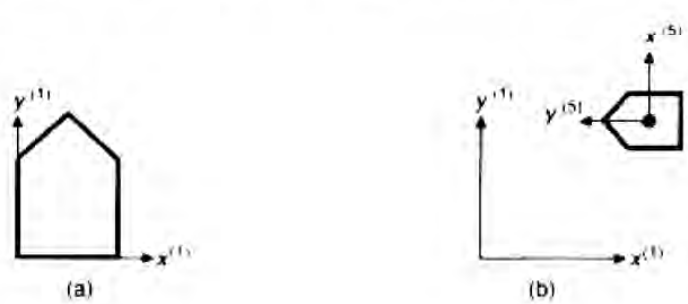


Fig. 5.25 The original house (a) in its coordinate system and the transformed house (b) in its coordinate system with respect to the original coordinate system.

An important question related to changing coordinate systems is changing transformations. Suppose $Q^{(j)}$ is a transformation in coordinate system j. It might, for example, be one of the composite transformations derived in previous sections. Suppose we wanted to find the transformation $Q^{(i)}$ in coordinate system i that could be applied to points $P^{(i)}$ in system i and produce exactly the same results as though $Q^{(j)}$ were applied to the corresponding points $P^{(j)}$ in system j. This equality is represented by $Q^{(j)} \cdot P^{(j)} = M_{i \leftarrow j} \cdot Q^{(j)} \cdot P^{(j)}$. Substituting $P^{(i)} = M_{i \leftarrow j} \cdot P^{(j)}$, this expression becomes $Q^{(i)} \cdot M_{i \leftarrow j} \cdot P^{(j)} = M_{i \leftarrow j} \cdot Q^{(j)} \cdot P^{(j)}$. Simplifying, we have $Q^{(i)} = M_{i \leftarrow j} \cdot Q^{(j)} \cdot M_{i \leftarrow j}^{-1}$.

The change-of-coordinate-system point of view is useful when additional information for subobjects is specified in the latters' own local coordinate systems. For example, if the front wheel of the tricycle in Fig. 5.26 is made to rotate about its z_{wh} coordinate, all wheels must be rotated appropriately, and we need to know how the tricycle as a whole moves in the world-coordinate system. This problem is complex because several successive changes of coordinate systems occur. First, the tricycle and front-wheel coordinate systems have initial positions in the world-coordinate system. As the bike moves forward, the front wheel rotates about the z axis of the wheel-coordinate system, and simultaneously the wheel- and tricycle-coordinate systems move relative to the world-coordinate system. The wheel- and tricycle-coordinate systems are related to the world-coordinate system by time-varying translations in x and z plus a rotation about y. The tricycle- and wheel-coordinate systems are related to each other by a time-varying rotation about y as the handlebars are turned. (The tricycle-coordinate system is fixed to the frame, not to the handlebars.)

To make the problem a bit easier, we assume that the wheel and tricycle axes are parallel to the world-coordinate axes and that the wheel moves in a straight line parallel to the world-coordinate x axis. As the wheel rotates by an angle α, a point P on the wheel rotates through the distance αr, where r is the radius of the wheel. Since the wheel is on the ground, the tricycle moves forward αr units. Therefore, the rim point P on the wheel moves and rotates with respect to the initial wheel-coordinate system with a net effect of translation by αr and rotation by α. Its new coordinates P' in the original wheel-coordinate system are thus

$$P'^{(wh)} = T(\alpha r, 0, 0) \cdot R_z(\alpha) \cdot P^{(wh)}, \tag{5.74}$$

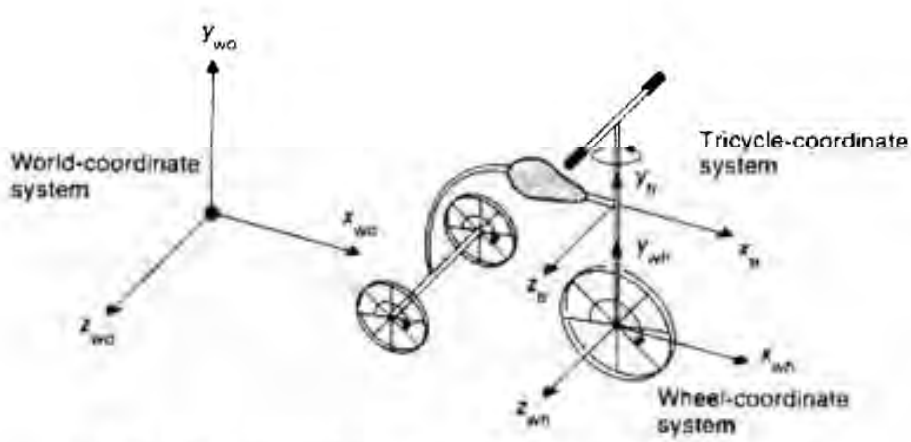


Fig. 5.26 A stylized tricycle with three coordinate systems.

and its coordinates in the new (translated) wheel-coordinate system are given by just the rotation

$$P'^{(wh')} = R_z(\alpha) \cdot P^{(wh)}. \tag{5.75}$$

To find the points $P^{(wo)}$ and $P'^{(wo)}$ in the world-coordinate system, we transform from the wheel to the world-coordinate system:

$$P^{(wo)} = M_{wo\leftarrow wh} \cdot P^{(wh)} = M_{wo\leftarrow tr} \cdot M_{tr\leftarrow wh} \cdot P^{(wh)}. \tag{5.76}$$

$M_{wo\leftarrow tr}$ and $M_{tr\leftarrow wh}$ are translations given by the initial positions of the tricycle and wheel. $P'^{(wo)}$ is computed with Eqs. (5.74) and (5.76):

$$P'^{(wo)} = M_{wo\leftarrow wh'} \cdot P'^{(wh')} = M_{wo\leftarrow wh} \cdot T(\alpha r, 0, 0) \cdot R_z(\alpha) \cdot P^{(wh)}. \tag{5.77}$$

Alternatively, we recognize that $M_{wo\leftarrow wh'}$ has been changed to $M_{wo\leftarrow wh}$ by the translation of the wheel-coordinate system, and get the same result as Eq. (5.77), but in a different way:

$$P'^{(wo)} = M_{wo\leftarrow wh'} \cdot P'^{(wh')} = (M_{wo\leftarrow wh} \cdot M_{wh\leftarrow wh'}) \cdot (R_z(\alpha) \cdot P^{wh}). \tag{5.78}$$

In general, then, we derive the new $M_{wo\leftarrow wh'}$ and $M_{tr'\leftarrow wh'}$ from their previous values by applying the appropriate transformations from the equations of motion of the tricycle parts. We then apply these updated transformations to updated points in local coordinate systems and derive the equivalent points in world-coordinate systems. We leave to the reader the problem of turning the tricycle's front wheel to change direction and of computing rotation angles for the rear wheels, using the wheels' radius and the tricycle's trajectory.

EXERCISES

5.1 Prove that we can transform a line by transforming its endpoints and then constructing a new line between the transformed endpoints.

5.2 Prove that two successive 2D rotations are additive: $R(\theta_1) \cdot R(\theta_2) = R(\theta_1 + \theta_2)$.

5.3 Prove that 2D rotation and scaling commute if $s_x = s_y$ or if $\theta = n\pi$ for integral n, and that otherwise they do not.

5.4 Find an expression relating the accumulated error in Eq. (5.37) to θ and the number of incremental rotations performed. Do the same for Eq. (5.38).

5.5 Write a program for your favorite computer to perform 2D incremental rotation. How much time is needed per endpoint? Compare this value to the time needed per endpoint for absolute 2D rotation.

5.6 A drawing consisting of N endpoints is to be rotated dynamically about a single axis. Multiplication on your computer takes time t_m; addition takes time t_a. Write expressions for the time needed to rotate the N points by using Eqs. (5.37), (5.38), and (5.7). Ignore control steps. Now evaluate the expressions with N as a variable, using the actual instruction times for your computer.

5.7 Apply the transformations developed in Section 5.7 to the points P_1, P_2, and P_3 to verify that these points transform as intended.

5.8 Rework Section 5.7, assuming that $|P_1P_2| = 1$, $|P_1P_3| = 1$ and that direction cosines of P_1P_2 and P_1P_3 are given (direction cosines of a line are the cosines of the angles between the line and the x, y, and z axes). For a line from the origin to (x, y, z), the direction cosines are $(x/d, y/d, z/d)$, where d is the length of the line.

5.9 Another reason that homogeneous coordinates are attractive is that 3D points at infinity in Cartesian coordinates can be represented explicitly in homogeneous coordinates. How can this be done?

5.10 Show that Eqs. (5.61) and (5.66) are equivalent.

5.11 Given a unit cube with one corner at (0, 0, 0) and the opposite corner at (1, 1, 1), derive the transformations necessary to rotate the cube by θ degrees about the main diagonal (from (0, 0, 0) to (1, 1, 1)) in the counterclockwise direction when looking along the diagonal toward the origin.

5.12 Suppose that the base of the window is rotated at an angle θ from the x axis, as in the Core System [GSPC79]. What is the window-to-viewport mapping? Verify your answer by applying the transformation to each corner of the window, to see that these corners are transformed to the appropriate corners of the viewport.

5.13 Consider a line from the origin of a right-handed coordinate system to the point $P(x, y, z)$. Find the transformation matrices needed to rotate the line into the positive z axis in three different ways, and show by algebraic manipulation that, in each case, the P does go to the z axis. For each method, calculate the sines and cosines of the angles of rotation.

a. Rotate about the x axis into the (x, y) plane, then rotate about the y axis into the z axis.
b. Rotate about the y axis into the (y, z) plane, then rotate about the x axis into the z axis.
c. Rotate about the z axis into the (x, z) plane, then rotate about the y axis into the z axis.

5.14 An object is to be scaled by a factor S in the direction whose direction cosines are (α, β, γ). Derive the transformation matrix.

5.15 Find the 4×4 transformation matrix that rotates by an angle θ about an arbitrary direction given by the direction vector $U = (u_x, u_y, u_z)$. Do this by composing the transformation matrix that rotates U into the z axis (call this M) with a rotation by $R_z(\theta)$, then composing this result with M^{-1}. The result should be

$$\begin{bmatrix} u_x^2 + \cos\theta(1 - u_x^2) & u_x u_y(1 - \cos\theta) - u_z\sin\theta & u_x u_z(1 - \cos\theta) + u_y\sin\theta & 0 \\ u_x u_y(1 - \cos\theta) + u_z\sin\theta & u_y^2 + \cos\theta(1 - u_y^2) & u_y u_z(1 - \cos\theta) - u_x\sin\theta & 0 \\ u_x u_z(1 - \cos\theta) - u_y\sin\theta & u_y u_z(1 - \cos\theta) + u_x\sin\theta & u_z^2 + \cos\theta(1 - u_z^2) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \quad (5.79)$$

Verify that, if U is a principal axis, the matrix reduces to R_x, R_y, or R_z. See [FAUX79] for a derivation based on vector operations. Note that negating both U and θ leave the result unchanged. Explain why this is true.

5.16 Prove the properties of $R(\theta)$ described at the end of Section 5.2.

5.17 Extend the incremental rotation discussed in Section 5.4 to 3D, forming a composite operation for rotation about an arbitrary axis.

5.18 Suppose the lowest rate at which an object can be rotated without being annoyingly slow is 360° over 30 seconds. Suppose also that, to be smooth, the rotation must be in steps of at most 4°. Use the results from Exercise 5.6 to determine how many points can be rotated using absolute rotation, and using incremental rotation.

5.19 Suppose that you are creating an interface to rotate an object by applying many incremental rotations, using "Spin X," "Spin Y" and "Spin Z" buttons. Each time one of these buttons is pressed, the current rotation matrix is replaced by its product with a matrix that rotates slightly around the specified axis. Although this idea is mathematically correct, in practice cumulative floating-point roundoff errors will result that will cause points to be transformed incorrectly. Show that by applying the Gram-Schmidt process to the columns of the matrix (see Section A.3.6) you can convert the new matrix back to an orthonormal matrix. Also explain why, if it is already orthonormal, applying this process will not change it.

16
Illumination and Shading

In this chapter, we discuss how to shade surfaces based on the position, orientation, and characteristics of the surfaces and the light sources illuminating them. We develop a number of different *illumination models* that express the factors determining a surface's color at a given point. Illumination models are also frequently called *lighting models* or *shading models*. Here, however, we reserve the term *shading model* for the broader framework in which an illumination model fits. The shading model determines when the illumination model is applied and what arguments it will receive. For example, some shading models invoke an illumination model for every pixel in the image, whereas others invoke an illumination model for only some pixels, and shade the remaining pixels by interpolation.

When we compared the accuracy with which the visible-surface calculations of the previous chapter are performed, we distinguished between algorithms that use the actual object geometry and those that use polyhedral approximations, between object-precision and image-precision algorithms, and between image-precision algorithms that take one point sample per pixel and those that use better filters. In all cases, however, the single criterion for determining the direct visibility of an object at a pixel is whether something lies between the object and the observer along the projector through the pixel. In contrast, the interaction between lights and surfaces is a good deal more complex. Graphics researchers have often approximated the underlying rules of optics and thermal radiation, either to simplify computation or because more accurate models were not known in the graphics community. Consequently, many of the illumination and shading models traditionally used in computer graphics include a multitude of kludges, "hacks," and simplifications that have no firm grounding in theory, but that work well in practice. The

ATI Ex. 2003
IPR2023-00564
Page 64 of 241

first part of this chapter covers these simple models, which are still in common use because they can produce attractive and useful results with minimal computation.

We begin, in Section 16.1, with a discussion of simple illumination models that take into account an individual point on a surface and the light sources directly illuminating it. We first develop illumination models for monochromatic surfaces and lights, and then show how the computations can be generalized to handle the color systems discussed in Chapter 13. Section 16.2 describes the most common shading models that are used with these illumination models. In Section 16.3, we expand these models to simulate textured surfaces.

Modeling refraction, reflection, and shadows requires additional computation that is very similar to, and often is integrated with, hidden-surface elimination. Indeed, these effects occur because some of the "hidden surfaces" are not really hidden at all—they are seen through, reflected from, or cast shadows on the surface being shaded! Sections 16.4 through 16.6 discuss how to model these effects. We next introduce, in Section 16.7, illumination models that more accurately characterize how an individual surface interacts with the light sources directly illuminating it. This is followed by coverage of additional ways to generate more realistic images, in Section 16.8 through 16.10.

Sections 16.11 through 16.13 describe *global illumination models* that attempt to take into account the interchange of light between all surfaces: recursive ray tracing and radiosity methods. Recursive ray tracing extends the visible-surface ray-tracing algorithm introduced in the previous chapter to interleave the determination of visibility, illumination, and shading at each pixel. Radiosity methods model the energy equilibrium in a system of surfaces; they determine the illumination of a set of sample points in the environment in a view-independent fashion before visible-surface determination is performed from the desired viewpoint. More detailed treatments of many of the illumination and shading models covered here may be found in [GLAS89; HALL89].

Finally, in Section 16.14, we look at several different graphics pipelines that integrate the rasterization techniques discussed in this and the previous chapters. We examine some ways to implement these capabilities to produce systems that are both efficient and extensible.

16.1 ILLUMINATION MODELS

16.1.1 Ambient Light

Perhaps the simplest illumination model possible is that used implicitly in this book's earliest chapters: Each object is displayed using an intensity intrinsic to it. We can think of this model, which has no external light source, as describing a rather unrealistic world of nonreflective, self-luminous objects. Each object appears as a monochromatic silhouette, unless its individual parts, such as the polygons of a polyhedron, are given different shades when the object is created. Color Plate II.28 demonstrates this effect.

An illumination model can be expressed by an *illumination equation* in variables associated with the point on the object being shaded. The illumination equation that expresses this simple model is

$$I = k_a \tag{16.1}$$

where I is the resulting intensity and the coefficient k_i is the object's intrinsic intensity. Since this illumination equation contains no terms that depend on the position of the point being shaded, we can evaluate it once for each object. The process of evaluating the illumination equation at one or more points on an object is often referred to as *lighting* the object.

Now imagine, instead of self-luminosity, that there is a diffuse, nondirectional source of light, the product of multiple reflections of light from the many surfaces present in the environment. This is known as *ambient* light. If we assume that ambient light impinges equally on all surfaces from all directions, then our illumination equation becomes

$$I = I_a k_a. \tag{16.2}$$

I_a is the intensity of the ambient light, assumed to be constant for all objects. The amount of ambient light reflected from an object's surface is determined by k_a, the *ambient-reflection coefficient*, which ranges from 0 to 1. The ambient-reflection coefficient is a *material property*. Along with the other material properties that we will discuss, it may be thought of as characterizing the material from which the surface is made. Like some of the other properties, the ambient-reflection coefficient is an empirical convenience and does not correspond directly to any physical property of real materials. Furthermore, ambient light by itself is not of much interest. As we see later, it is used to account for all the complex ways in which light can reach an object that are not otherwise addressed by the illumination equation. Color Plate II.28 also demonstrates illumination by ambient light.

16.1.2 Diffuse Reflection

Although objects illuminated by ambient light are more or less brightly lit in direct proportion to the ambient intensity, they are still uniformly illuminated across their surfaces. Now consider illuminating an object by a *point light source*, whose rays emanate uniformly in all directions from a single point. The object's brightness varies from one part to another, depending on the direction of and distance to the light source.

Lambertian reflection. Dull, matte surfaces, such as chalk, exhibit *diffuse reflection*, also known as *Lambertian reflection*. These surfaces appear equally bright from all viewing angles because they reflect light with equal intensity in all directions. For a given surface, the brightness depends only on the angle θ between the direction $\overline{L}$ to the light source and the surface normal $\overline{N}$ of Fig. 16.1. Let us examine why this occurs. There are two factors at work here. First, Fig. 16.2 shows that a beam that intercepts a surface covers an area whose size is inversely proportional to the cosine of the angle θ that the beam makes with $\overline{N}$. If the

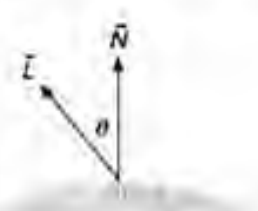


Fig. 16.1 Diffuse reflection.

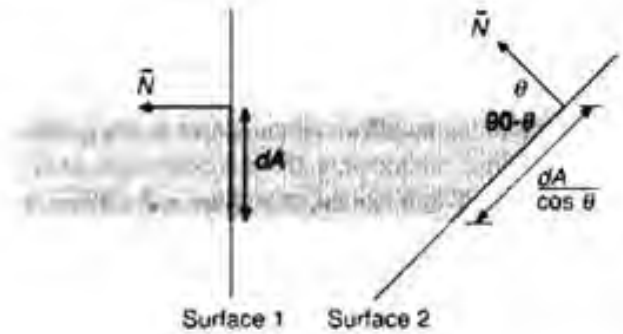


Fig. 16.2 Beam (shown in 2D cross-section) of infinitesimal cross-sectional area dA at angle of incidence θ intercepts area of dA / cos θ.

beam has an infinitesimally small cross-sectional differential area dA, then the beam intercepts an area dA / cos θ on the surface. Thus, for an incident light beam, the amount of light energy that falls on dA is proportional to cos θ. This is true for any surface, independent of its material.

Second, we must consider the amount of light seen by the viewer. Lambertian surfaces have the property, often known as Lambert's law, that the amount of light reflected from a unit differential area dA toward the viewer is directly proportional to the cosine of the angle between the direction to the viewer and $\bar{N}$. Since the amount of surface area seen is inversely proportional to the cosine of this angle, these two factors cancel out. For example, as the viewing angle increases, the viewer sees more surface area, but the amount of light reflected at that angle per unit area of surface is proportionally less. Thus, for Lambertian surfaces, the amount of light seen by the viewer is independent of the viewer's direction and is proportional only to cos θ, the angle of incidence of the light.

The diffuse illumination equation is

$$I = I_p k_d \cos\theta. \tag{16.3}$$

I_p is the point light source's intensity; the material's *diffuse-reflection coefficient* k_d is a constant between 0 and 1 and varies from one material to another. The angle θ must be between 0° and 90° if the light source is to have any direct effect on the point being shaded. This means that we are treating the surface as *self-occluding*, so that light cast from behind a point on the surface does not illuminate it. Rather than include a max(cos θ, 0) term explicitly here and in the following equations, we assume that θ lies within the legal range. When we want to light self-occluding surfaces, we can use abs(cos θ) to invert their surface normals. This causes both sides of the surface to be treated alike, as if the surface were lit by two opposing lights.

Assuming that the vectors $\bar{N}$ and $\bar{L}$ have been normalized (see Appendix), we can rewrite Eq. (16.3) by using the dot product:

$$I = I_p k_d (\bar{N} \cdot \bar{L}). \tag{16.4}$$

The surface normal $\bar{N}$ can be calculated using the methods discussed in Chapter 11. If polygon normals are precomputed and transformed with the same matrix used for the polygon vertices, it is important that nonrigid modeling transformations, such as shears or

Fig. 16.3 Spheres shaded using a diffuse-reflection model (Eq. 16.4). For all spheres, $I_p = 1.0$. From left to right, $k_d = 0.4, 0.55, 0.7, 0.85, 1.0$. (By David Kurlander, Columbia University.)

differential scaling, not be performed; these transformations do not preserve angles and may cause some normals to be no longer perpendicular to their polygons. The proper method to transform normals when objects undergo arbitrary transformations is described in Section 5.6. In any case, the illumination equation must be evaluated in the WC system (or in any coordinate system isometric to it), since both the normalizing and perspective transformations will modify θ.

If a point light source is sufficiently distant from the objects being shaded, it makes essentially the same angle with all surfaces sharing the same surface normal. In this case, the light is called a *directional light source*, and $\overline{L}$ is a constant for the light source.

Figure 16.3 shows a series of pictures of a sphere illuminated by a single point source. The shading model calculated the intensity at each pixel at which the sphere was visible using the illumination model of Eq. (16.4). Objects illuminated in this way look harsh, as when a flashlight illuminates an object in an otherwise dark room. Therefore, an ambient term is commonly added to yield a more realistic illumination equation:

$$I = I_a k_a + I_p k_d (\overline{N} \cdot \overline{L}). \tag{16.5}$$

Equation (16.5) was used to produce Fig. 16.4.

Light-source attenuation. If the projections of two parallel surfaces of identical material, lit from the eye, overlap in an image, Eq. (16.5) will not distinguish where one surface leaves off and the other begins, no matter how different are their distances from the light source. To do this, we introduce a light-source attenuation factor, f_{att}, yielding

$$I = I_a k_a + f_{att} I_p k_d (\overline{N} \cdot \overline{L}). \tag{16.6}$$

Fig. 16.4 Spheres shaded using ambient and diffuse reflection (Eq. 16.5). For all spheres, $I_a = I_p = 1.0$, $k_d = 0.4$. From left to right, $k_a = 0.0, 0.15, 0.30, 0.45, 0.60$. (By David Kurlander, Columbia University.)

An obvious choice for f_{att} takes into account the fact that the energy from a point light source that reaches a given part of a surface falls off as the inverse square of d_L, the distance the light travels from the point source to the surface. In this case,

$$f_{att} = \frac{1}{d_L^2}. \tag{16.7}$$

In practice, however, this often does not work well. If the light is far away, $1 / d_L^2$ does not vary much; if it is very close, it varies widely, giving considerably different shades to surfaces with the same angle θ between $\bar{N}$ and $\bar{L}$. Although this behavior is correct for a point light source, the objects we see in real life typically are not illuminated by point sources and are not shaded using the simplified illumination models of computer graphics. To complicate matters, early graphics researchers often used a single point light source positioned right at the viewpoint. They expected f_{att} to approximate some of the effects of atmospheric attenuation between the viewer and the object (see Section 16.1.3), as well as the energy density falloff from the light to the object. A useful compromise, which allows a richer range of effects than simple square-law attenuation, is

$$f_{att} = \min\left(\frac{1}{c_1 + c_2 d_L + c_3 d_L^2}, 1\right). \tag{16.8}$$

Here c_1, c_2, and c_3 are user-defined constants associated with the light source. The constant c_1 keeps the denominator from becoming too small when the light is close, and the expression is clamped to a maximum of 1 to ensure that it always attenuates. Figure 16.5 uses this illumination model with different constants to show a range of effects.

Colored lights and surfaces. So far, we have described monochromatic lights and surfaces. Colored lights and surfaces are commonly treated by writing separate equations for each component of the color model. We represent an object's *diffuse color* by one value of O_d for each component. For example, the triple (O_{dR}, O_{dG}, O_{dB}) defines an object's diffuse red, green, and blue components in the RGB color system. In this case, the illuminating light's three primary components, I_{pR}, I_{pG}, and I_{pB}, are reflected in proportion to $k_d O_{dR}$, $k_d O_{dG}$, and $k_d O_{dB}$, respectively. Therefore, for the red component,

$$I_R = I_{aR} k_a O_{dR} + f_{att} I_{pR} k_d O_{dR} (\bar{N} \cdot \bar{L}). \tag{16.9}$$

Similar equations are used for I_G and I_B, the green and blue components. The use of a single coefficient to scale an expression in each of the equations allows the user to control the amount of ambient or diffuse reflection, without altering the proportions of its components. An alternative formulation that is more compact, but less convenient to control, uses a separate coefficient for each component; for example, substituting k_{aR} for $k_a O_{dR}$ and k_{dR} for $k_d O_{dR}$.

A simplifying assumption is made here that a three-component color model can completely model the interaction of light with objects. This assumption is wrong, as we discuss in Section 16.9, but it is easy to implement and often yields acceptable pictures. In theory, the illumination equation should be evaluated continuously over the spectral range being modeled; in practice, it is evaluated for some number of discrete spectral samples.

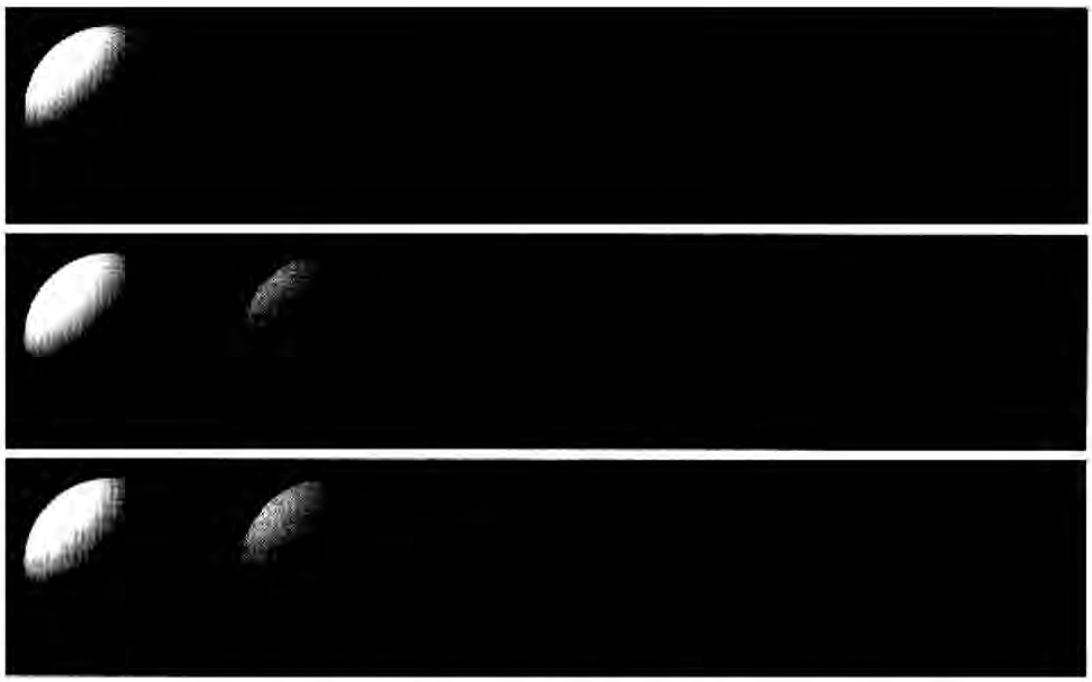


Fig. 16.5 Spheres shaded using ambient and diffuse reflection with a light-source-attenuation term (Eqs. 16.6 and 16.8). For all spheres, $I_a = I_p = 1.0$, $k_a = 0.1$, $k_d = 0.9$. From left to right, sphere's distance from light source is 1.0, 1.375, 1.75, 2.125, 2.5. Top row: $c_1 = c_2 = 0.0$, $c_3 = 1.0$ $(1/d_L^2)$. Middle row: $c_1 = c_2 = 0.25$, $c_3 = 0.5$. Bottom row: $c_1 = 0.0$, $c_2 = 1.0$, $c_3 = 0.0$ $(1/d_L)$. (By David Kurlander, Columbia University.)

Rather than restrict ourselves to a particular color model, we explicitly indicate those terms in an illumination equation that are wavelength-dependent by subscripting them with a λ. Thus, Eq. (16.9) becomes

$$I_\lambda = I_{a\lambda} k_a O_{d\lambda} + f_{att} I_{p\lambda} k_d O_{d\lambda} (\overline{N} \cdot \overline{L}). \tag{16.10}$$

16.1.3 Atmospheric Attenuation

To simulate the atmospheric attenuation from the object to the viewer, many systems provide *depth cueing*. In this technique, which originated with vector-graphics hardware, more distant objects are rendered with lower intensity than are closer ones. The PHIGS+ standard recommends a depth-cueing approach that also makes it possible to approximate the shift in colors caused by the intervening atmosphere. Front and back depth-cue reference planes are defined in NPC; each of these planes is associated with a scale factor, s_f and s_b, respectively, that ranges between 0 and 1. The scale factors determine the blending of the original intensity with that of a depth-cue color, $I_{dc\lambda}$. The goal is to modify a previously computed I_λ to yield the depth-cued value I'_λ that is displayed. Given z_o, the object's z coordinate, a scale factor s_o is derived that will be used to interpolate between I_λ and $I_{dc\lambda}$, to determine

$$I'_\lambda = s_o I_\lambda + (1 - s_o) I_{dc\lambda}. \tag{16.11}$$

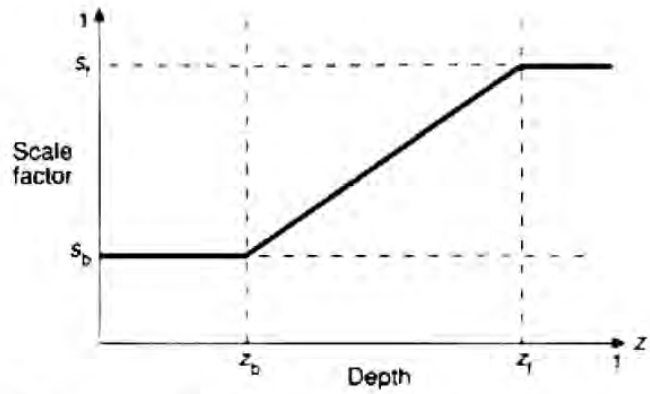


Fig. 16.6 Computing the scale factor for atmospheric attenuation.

If z_o is in front of the front depth-cue reference plane's z coordinate z_f, then $s_o = s_f$. If z_o is behind the back depth-cue reference plane's z coordinate z_b, then $s_o = s_b$. Finally, if z_o is between the two planes, then

$$s_o = s_b + \frac{(z_o - z_b)(s_f - s_b)}{z_f - z_b}. \tag{16.12}$$

The relationship between s_o and z_o is shown in Fig. 16.6. Figure 16.7 shows spheres shaded with depth cueing. To avoid complicating the equations, we ignore depth cueing as we develop the illumination model further. More realistic ways to model atmospheric effects are discussed in Section 20.8.2.

16.1.4 Specular Reflection

Specular reflection can be observed on any shiny surface. Illuminate an apple with a bright white light: The highlight is caused by specular reflection, whereas the light reflected from the rest of the apple is the result of diffuse reflection. Also note that, at the highlight, the apple appears to be not red, but white, the color of the incident light. Objects such as waxed apples or shiny plastics have a transparent surface; plastics, for example, are typically composed of pigment particles embedded in a transparent material. Light specularly reflected from the colorless surface has much the same color as that of the light source.

Fig. 16.7 Spheres shaded using depth cueing (Eqs. 16.5, 16.11, and 16.12). Distance from light is constant. For all spheres, $I_a = I_p = 1.0$, $k_a = 0.1$, $k_d = 0.9$, $z_f = 1.0$, $z_b = 0.0$, $s_f = 1.0$, $s_b = 0.1$, radius = 0.09. From left to right, z at front of sphere is 1.0, 0.77, 0.55, 0.32, 0.09. (By David Kurlander, Columbia University.)

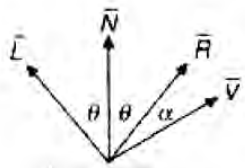


Fig. 16.8 Specular reflection.

Now move your head and notice how the highlight also moves. It does so because shiny surfaces reflect light unequally in different directions: on a perfectly shiny surface, such as a perfect mirror, light is reflected *only* in the direction of reflection $\bar{R}$, which is $\bar{L}$ mirrored about $\bar{N}$. Thus the viewer can see specularly reflected light from a mirror only when the angle α in Fig. 16.8 is zero; α is the angle between $\bar{R}$ and the direction to the viewpoint $\bar{V}$.

The Phong illumination model. Phong Bui-Tuong [BUIT75] developed a popular illumination model for nonperfect reflectors, such as the apple. It assumes that maximum specular reflectance occurs when α is zero and falls off sharply as α increases. This rapid falloff is approximated by $\cos^n \alpha$, where n is the material's *specular-reflection exponent*. Values of n typically vary from 1 to several hundred, depending on the surface material being simulated. A value of 1 provides a broad, gentle falloff, whereas higher values simulate a sharp, focused highlight (Fig. 16.9). For a perfect reflector, n would be infinite. As before, we treat a negative value of $\cos \alpha$ as zero. Phong's illumination model is based on earlier work by researchers such as Warnock [WARN69], who used a $\cos^n \theta$ term to model specular reflection with the light at the viewpoint. Phong, however, was the first to account for viewers and lights at arbitrary positions.

The amount of incident light specularly reflected depends on the angle of incidence θ. If $W(\theta)$ is the fraction of specularly reflected light, then Phong's model is

$$I_\lambda = I_{a\lambda} k_a O_{d\lambda} + f_{att} I_{p\lambda} [k_d O_{d\lambda} \cos \theta + W(\theta) \cos^n \alpha]. \tag{16.13}$$

If the direction of reflection $\bar{R}$, and the viewpoint direction $\bar{V}$ are normalized, then $\cos \alpha = \bar{R} \cdot \bar{V}$. In addition, $W(\theta)$ is typically set to a constant k_s, the material's *specular-reflection coefficient*, which ranges between 0 and 1. The value of k_s is selected experimentally to

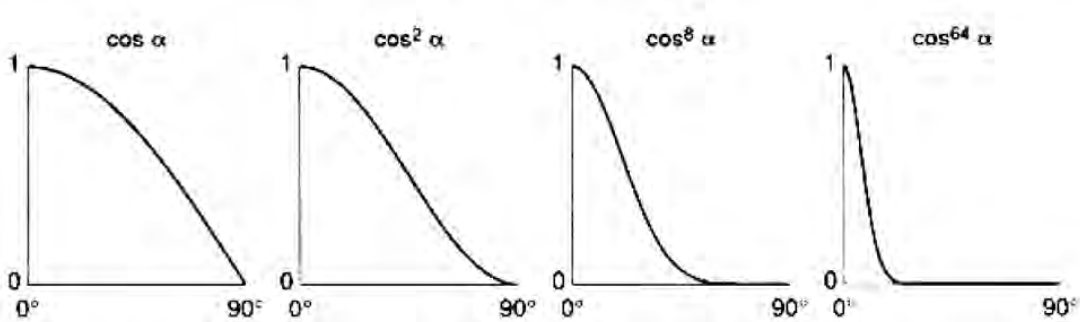


Fig. 16.9 Different values of $\cos^n \alpha$ used in the Phong illumination model.

produce aesthetically pleasing results. Then, Eq. (16.13) can be rewritten as

$$I_\lambda = I_{a\lambda} k_a O_{d\lambda} + f_{att} I_{p\lambda} [k_d O_{d\lambda} (\bar{N} \cdot \bar{L}) + k_s (\bar{R} \cdot \bar{V})^n]. \tag{16.14}$$

Note that the color of the specular component in Phong's illumination model is *not* dependent on any material property; thus, this model does a good job of modeling specular reflections from plastic surfaces. As we discuss in Section 16.7, specular reflection is affected by the properties of the surface itself, and, in general, may have a different color than diffuse reflection when the surface is a composite of several materials. We can accommodate this effect to a first approximation by modifying Eq. (16.14) to yield

$$I_\lambda = I_{a\lambda} k_a O_{d\lambda} + f_{att} I_{p\lambda} [k_d O_{d\lambda} (\bar{N} \cdot \bar{L}) + k_s O_{s\lambda} (\bar{R} \cdot \bar{V})^n], \tag{16.15}$$

where $O_{s\lambda}$ is the object's *specular color*. Figure 16.10 shows a sphere illuminated using Eq. (16.14) with different values of k_s and n.

Calculating the reflection vector. Calculating $\bar{R}$ requires mirroring $\bar{L}$ about $\bar{N}$. As shown in Fig. 16.11, this can be accomplished with some simple geometry. Since $\bar{N}$ and $\bar{L}$ are normalized, the projection of $\bar{L}$ onto $\bar{N}$ is $\bar{N} \cos \theta$. Note that $\bar{R} = \bar{N} \cos \theta + \bar{S}$, where $|\bar{S}|$ is $\sin \theta$. But, by vector subtraction and congruent triangles, $\bar{S}$ is just $\bar{N} \cos \theta - \bar{L}$. Therefore, $\bar{R} = 2\bar{N} \cos \theta - \bar{L}$. Substituting $\bar{N} \cdot \bar{L}$ for $\cos \theta$ and $\bar{R} \cdot \bar{V}$ for $\cos \alpha$ yields

$$\bar{R} = 2\bar{N} (\bar{N} \cdot \bar{L}) - \bar{L}, \tag{16.16}$$

$$\bar{R} \cdot \bar{V} = (2\bar{N} (\bar{N} \cdot \bar{L}) - \bar{L}) \cdot \bar{V}. \tag{16.17}$$

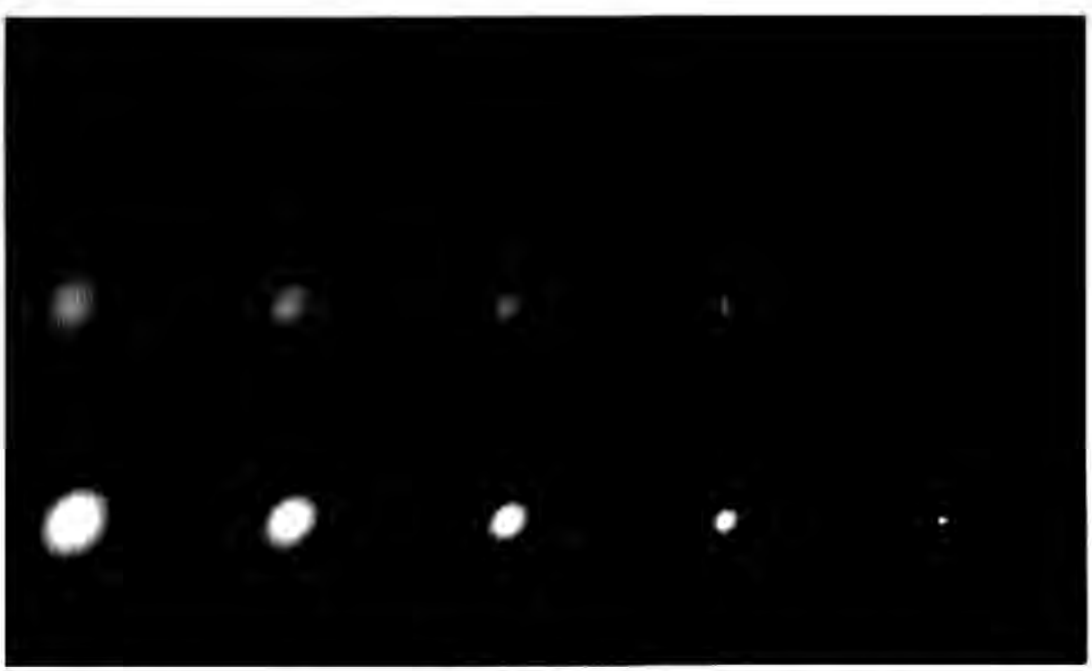


Fig. 16.10 Spheres shaded using Phong's illumination model (Eq. 16.14) and different values of k_s and n. For all spheres, $I_a = I_p = 1.0$, $k_a = 0.1$, $k_d = 0.45$. From left to right, n = 3.0, 5.0, 10.0, 27.0, 200.0. From top to bottom, k_s = 0.1, 0.25, 0.5. (By David Kurlander, Columbia University.)

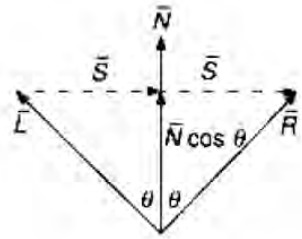


Fig. 16.11 Calculating the reflection vector.

If the light source is at infinity, $\bar{N} \cdot \bar{L}$ is constant for a given polygon, whereas $\bar{R} \cdot \bar{V}$ varies across the polygon. For curved surfaces or for a light source not at infinity, both $\bar{N} \cdot \bar{L}$ and $\bar{R} \cdot \bar{V}$ vary across the surface.

The halfway vector. An alternative formulation of Phong's illumination model uses the *halfway vector* $\bar{H}$, so called because its direction is halfway between the directions of the light source and the viewer, as shown in Fig. 16.12. $\bar{H}$ is also known as the direction of maximum highlights. If the surface were oriented so that its normal were in the same direction as $\bar{H}$, the viewer would see the brightest specular highlight, since $\bar{R}$ and $\bar{V}$ would also point in the same direction. The new specular-reflection term can be expressed as $(\bar{N} \cdot \bar{H})^n$, where $\bar{H} = (\bar{L} + \bar{V}) / |\bar{L} + \bar{V}|$. When the light source and the viewer are both at infinity, then the use of $\bar{N} \cdot \bar{H}$ offers a computational advantage, since $\bar{H}$ is constant. Note that β, the angle between $\bar{N}$ and $\bar{H}$, is not equal to α, the angle between $\bar{R}$ and $\bar{V}$, so the same specular exponent n produces different results in the two formulations (see Exercise 16.1). Although using a $\cos^n$ term allows the generation of recognizably glossy surfaces, you should remember that it is based on empirical observation, not on a theoretical model of the specular-reflection process.

16.1.5 Improving the Point-Light-Source Model

Real light sources do not radiate equally in all directions. Warn [WARN83] has developed easily implemented lighting controls that can be added to any illumination equation to model some of the directionality of the lights used by photographers. In Phong's model, a point light source has only an intensity and a position. In Warn's model, a light L is modeled by a point on a hypothetical specular reflecting surface, as shown in Fig. 16.13. This surface is illuminated by a point light source L' in the direction $\bar{L}'$. Assume that $\bar{L}'$ is normal to the hypothetical reflecting surface. Then, we can use the Phong illumination

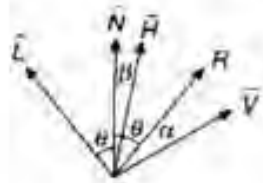


Fig. 16.12 $\bar{H}$, the halfway vector, is halfway between the direction of the light source and the viewer.

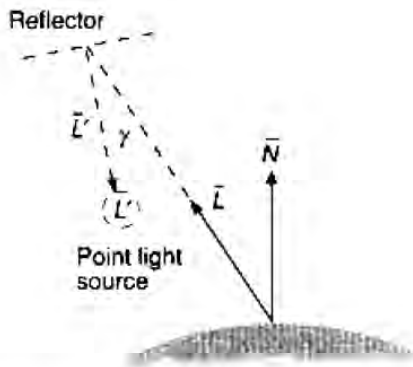


Fig. 16.13 Warn's lighting model. A light is modeled as the specular reflection from a single point illuminated by a point light source.

equation to determine the intensity of L at a point on the object in terms of the angle γ between $\bar{L}$ and $\bar{L}'$. If we further assume that the reflector reflects only specular light and has a specular coefficient of 1, then the light's intensity at a point on the object is

$$I_{L'\lambda} \cos^p \gamma, \tag{16.18}$$

where $I_{L'\lambda}$ is the intensity of the hypothetical point light source, p is the reflector's specular exponent, and γ is the angle between $-\bar{L}$ and the hypothetical surface's normal, $\bar{L}'$, which is the direction to L'. Equation (16.18) models a symmetric directed light source whose axis of symmetry is $\bar{L}'$, the direction in which the light may be thought of as pointing. Using dot products, we can write Eq.(16.18) as

$$I_{L'\lambda} (-\bar{L} \cdot \bar{L}')^p. \tag{16.19}$$

Once again, we treat a negative dot product as zero. Equation (16.19) can thus be substituted for the light-source intensity $I_{p\lambda}$ in the formulation of Eq. (16.15) or any other illumination equation. Contrast the intensity distribution of the uniformly radiating point source with the $\cos^p$ distribution of the Warn light source in Fig. 16.14. Each distribution is plotted in cross-section, showing intensity as a function of angular direction around the light's axis in polar coordinates. $\bar{L}'$ is shown as an arrow. These plots are called *goniometric diagrams*. The larger the value of p, the more the light is concentrated along $\bar{L}'$. Thus, a

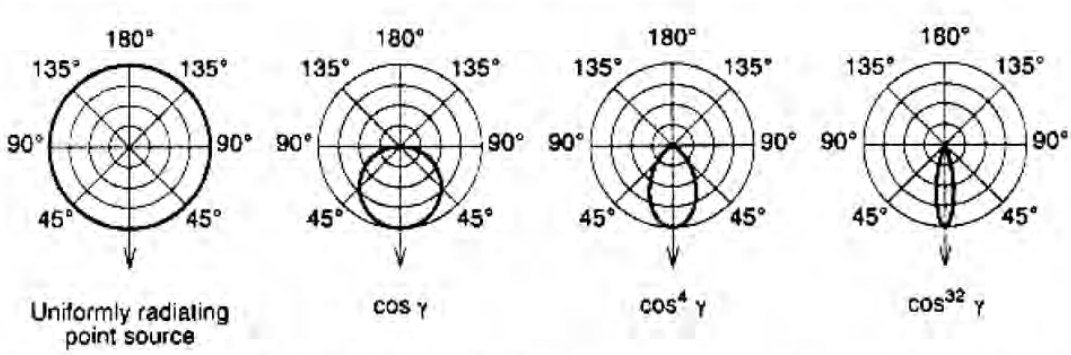


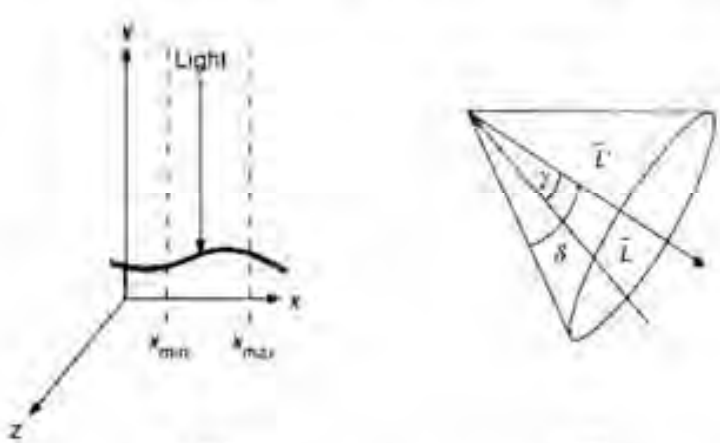
Fig. 16.14 Intensity distributions for uniformly radiating point source and Warn light source with different values of p.

(a) (b) (c) (d) (e)

Fig. 16.15 Cube and plane illuminated using Warn lighting controls. (a) Uniformly radiating point source (or $p = 0$). (b) $p = 4$. (c) $p = 32$. (d) Flaps. (e) Cone with $\delta = 18°$. (By David Kurlander, Columbia University.)

large value of p can simulate a highly directional spotlight, whereas a small value of p can simulate a more diffuse floodlight. If p is 0, then the light acts like a uniformly radiating point source. Figure 16.15(a–c) shows the effects of different values of p. Verbeck [VERB84] and Nishita et al. [NISH85b] have modeled point light sources with more complex irregular intensity and spectral distributions. In general, however, once we determine a point light source's intensity as seen from a particular direction, this value can be used in any illumination equation.

To restrict a light's effects to a limited area of the scene, Warn implemented *flaps* and *cones*. Flaps, modeled loosely after the "barn doors" found on professional photographic lights, confine the effects of the light to a designated range in x, y, and z world coordinates. Each light has six flaps, corresponding to user-specified minimum and maximum values in each coordinate. Each flap also has a flag indicating whether it is on or off. When a point's shade is determined, the illumination model is evaluated for a light only if the point's coordinates are within the range specified by the minimum and maximum coordinates of those flaps that are on. For example, if $\bar{L}'$ is parallel to the y axis, then the x and z flaps can sharply restrict the light's effects, much like the photographic light's barn doors. Figure 16.16(a) shows the use of the x flaps in this situation. The y flaps can also be used here to



(a) (b)

Fig. 16.16 The Warn intensity distribution may be restricted with (a) flaps and (b) cones.

restrict the light in a way that has no physical counterpart, allowing only objects within a specified range of distances from the light to be illuminated. In Fig. 16.15(d) the cube is aligned with the coordinate system, so two pairs of flaps can produce the effects shown.

Warn makes it possible to create a sharply delineated spotlight through the use of a cone whose apex is at the light source and whose axis lies along $\overline{L}'$. As shown in Fig. 16.16(b), a cone with a generating angle of δ may be used to restrict the light source's effects by evaluating the illumination model only when $\gamma < \delta$ (or when $\cos\gamma > \cos\delta$, since $\cos\gamma$ has already been calculated). The PHIGS+ illumination model includes the Warn $\cos^e\gamma$ term and cone angle δ. Figure 16.15(e) demonstrates the use of a cone to restrict the light of Fig. 16.15(c). Color Plate II.17 shows a car rendered with Warn's lighting controls.

16.1.6 Multiple Light Sources

If there are m light sources, then the terms for each light source are summed:

$$I_\lambda = I_{a\lambda} k_a O_{d\lambda} + \sum_{1 \le i \le m} f_{att_i} I_{p\lambda_i} [k_d O_{d\lambda} (\overline{N} \cdot \overline{L}_i) + k_s O_{s\lambda} (\overline{R}_i \cdot \overline{V})^n]. \qquad (16.20)$$

The summation harbors a new possibility for error in that I_λ can now exceed the maximum displayable pixel value. (Although this can also happen for a single light, we can easily avoid it by an appropriate choice of f_{att} and the material coefficients.) Several approaches can be used to avoid overflow. The simplest is to clamp each I_λ individually to its maximum value. Another approach considers all of a pixel's I_λ values together. If at least one is too big, each is divided by the largest to maintain the hue and saturation at the expense of the value. If all the pixel values can be computed before display, image-processing transformations can be applied to the entire picture to bring the values within the desired range. Hall [HALL89] discusses the tradeoffs of these and other techniques.

16.2 SHADING MODELS FOR POLYGONS

It should be clear that we can shade any surface by calculating the surface normal at each visible point and applying the desired illumination model at that point. Unfortunately, this brute-force shading model is expensive. In this section, we describe more efficient shading models for surfaces defined by polygons and polygon meshes.

16.2.1 Constant Shading

The simplest shading model for a polygon is *constant shading*, also known as *faceted shading* or *flat shading*. This approach applies an illumination model once to determine a single intensity value that is then used to shade an entire polygon. In essence, we are sampling the value of the illumination equation once for each polygon, and holding the value across the polygon to reconstruct the polygon's shade. This approach is valid if several assumptions are true:

1. The light source is at infinity, so $\overline{N} \cdot \overline{L}$ is constant across the polygon face
2. The viewer is at infinity, so $\overline{N} \cdot \overline{V}$ is constant across the polygon face
3. The polygon represents the actual surface being modeled, and is not an approximation to a curved surface.

If a visible-surface algorithm is used that outputs a list of polygons, such as one of the list-priority algorithms, constant shading can take advantage of the ubiquitous single-color 2D polygon primitive.

If either of the first two assumptions is wrong, then, if we are to use constant shading, we need some method to determine a single value for each of $\overline{L}$ and $\overline{V}$. For example, values may be calculated for the center of the polygon, or for the polygon's first vertex. Of course, constant shading does not produce the variations in shade across the polygon that should occur in this situation.

16.2.2 Interpolated Shading

As an alternative to evaluating the illumination equation at each point on the polygon, Wylie, Romney, Evans, and Erdahl [WYLI67] pioneered the use of *interpolated shading*, in which shading information is linearly interpolated across a triangle from values determined for its vertices. Gouraud [GOUR71] generalized this technique to arbitrary polygons. This is particularly easy for a scan-line algorithm that already interpolates the z value across a span from interpolated z values computed for the span's endpoints. For increased efficiency, a difference equation may be used, like that developed in Section 15.4 to determine the z value at each pixel. Although z interpolation is physically correct (assuming that the polygon is planar), note that interpolated shading is not, since it only approximates evaluating the illumination model at each point on the polygon.

Our final assumption, that the polygon accurately represents the surface being modeled, is most often the one that is incorrect, which has a much more substantial effect on the resulting image than does the failure of the other two assumptions. Many objects are curved, rather than polyhedral, yet representing them as a polygon mesh allows the use of efficient polygon visible-surface algorithms. We discuss next how to render a polygon mesh so that it looks as much as possible like a curved surface.

16.2.3 Polygon Mesh Shading

Suppose that we wish to approximate a curved surface by a polygonal mesh. If each polygonal facet in the mesh is shaded individually, it is easily distinguished from neighbors whose orientation is different, producing a "faceted" appearance, as shown in Color Plate II.29. This is true if the polygons are rendered using constant shading, interpolated shading, or even per-pixel illumination calculations, because two adjacent polygons of different orientation have different intensities along their borders. The simple solution of using a finer mesh turns out to be surprisingly ineffective, because the perceived difference in shading between adjacent facets is accentuated by the Mach band effect (discovered by Mach in 1865 and described in detail in [RATL72]), which exaggerates the intensity change at any edge where there is a discontinuity in magnitude or slope of intensity. At the border between two facets, the dark facet looks darker and the light facet looks lighter. Figure 16.17 shows, for two separate cases, the actual and perceived changes in intensity along a surface.

Mach banding is caused by *lateral inhibition* of the receptors in the eye. The more light a receptor receives, the more that receptor inhibits the response of the receptors adjacent to

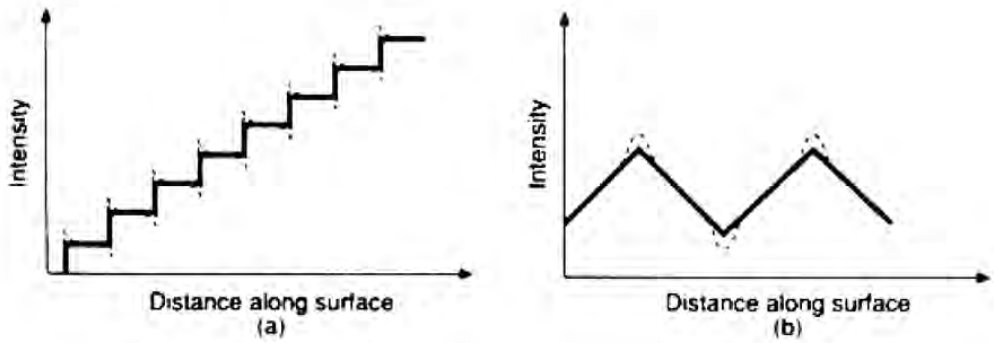


Fig. 16.17 Actual and perceived intensities in the Mach band effect. Dashed lines are perceived intensity; solid lines are actual intensity.

it. The response of a receptor to light is inhibited by its adjacent receptors in inverse relation to the distance to the adjacent receptor. Receptors directly on the brighter side of an intensity change have a stronger response than do those on the brighter side that are farther from the edge, because they receive less inhibition from their neighbors on the darker side. Similarly, receptors immediately to the darker side of an intensity change have a weaker response than do those farther into the darker area, because they receive more inhibition from their neighbors on the brighter side. The Mach band effect is quite evident in Color Plate II.29, especially between adjacent polygons that are close in color.

The polygon-shading models we have described determine the shade of each polygon individually. Two basic shading models for polygon meshes take advantage of the information provided by adjacent polygons to simulate a smooth surface. In order of increasing complexity (and realistic effect), they are known as Gouraud shading and Phong shading, after the researchers who developed them. Current 3D graphics workstations typically support one or both of these approaches through a combination of hardware and firmware.

16.2.4 Gouraud Shading

Gouraud shading [GOUR71], also called *intensity interpolation shading* or *color interpolation shading*, eliminates intensity discontinuities. Color Plate II.30 uses Gouraud shading. Although most of the Mach banding of Color Plate II.29 is no longer visible in Color Plate II.30, the bright ridges on objects such as the torus and cone are Mach bands caused by a rapid, although not discontinuous, change in the slope of the intensity curve; Gouraud shading does not completely eliminate such intensity changes.

Gouraud shading extends the concept of interpolated shading applied to individual polygons by interpolating polygon vertex illumination values that take into account the surface being approximated. The Gouraud shading process requires that the normal be known for each vertex of the polygonal mesh. Gouraud was able to compute these *vertex normals* directly from an analytical description of the surface. Alternatively, if the vertex normals are not stored with the mesh and cannot be determined directly from the actual surface, then, Gouraud suggested, we can approximate them by averaging the surface

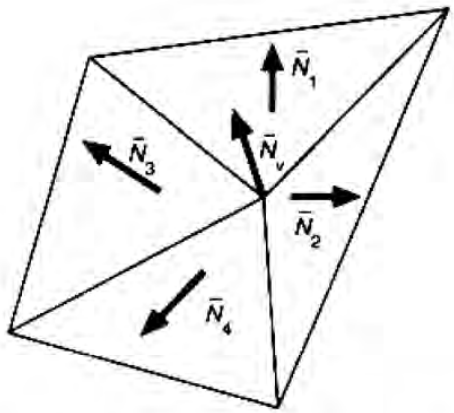
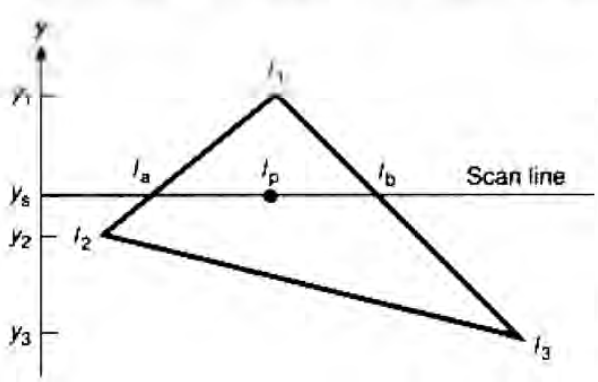


Fig. 16.18 Normalized polygon surface normals may be averaged to obtain vertex normals. Averaged normal $\bar{N}_v$ is $\sum_{1 \le i \le n} \bar{N}_i \,/\, |\sum_{1 \le i \le n} \bar{N}_i|$.

normals of all polygonal facets sharing each vertex (Fig. 16.18). If an edge is meant to be visible (as at the joint between a plane's wing and body), then we find two vertex normals, one for each side of the edge, by averaging the normals of polygons on each side of the edge separately. Normals were not averaged across the teapot's patch cracks in Color Plate II.30. (See caption to Color Plate II.21.)

The next step in Gouraud shading is to find *vertex intensities* by using the vertex normals with any desired illumination model. Finally, each polygon is shaded by linear interpolation of vertex intensities along each edge and then between edges along each scan line (Fig. 16.19) in the same way that we described interpolating z values in Section 15.4. The term *Gouraud shading* is often generalized to refer to intensity interpolation shading of even a single polygon in isolation, or to the interpolation of arbitrary colors associated with polygon vertices.

The interpolation along edges can easily be integrated with the scan-line visible-surface algorithm of Section 15.6. With each edge, we store for each color component the starting intensity and the change of intensity for each unit change in y. A visible span on a scan line is filled in by interpolating the intensity values of the two edges bounding the span. As in all linear-interpolation algorithms, a difference equation may be used for increased efficiency.



$$I_a = I_1 - (I_1 - I_2)\,\frac{y_1 - y_s}{y_1 - y_2}$$

$$I_b = I_1 - (I_1 - I_3)\,\frac{y_1 - y_s}{y_1 - y_3}$$

$$I_p = I_b - (I_b - I_a)\,\frac{x_b - x_p}{x_b - x_a}$$

Fig. 16.19 Intensity interpolation along polygon edges and scan lines.

16.2.5 Phong Shading

Phong shading [BUIT75], also known as *normal-vector interpolation shading*, interpolates the surface normal vector $\bar{N}$, rather than the intensity. Interpolation occurs across a polygon span on a scan line, between starting and ending normals for the span. These normals are themselves interpolated along polygon edges from vertex normals that are computed, if necessary, just as in Gouraud shading. The interpolation along edges can again be done by means of incremental calculations, with all three components of the normal vector being incremented from scan line to scan line. At each pixel along a scan line, the interpolated normal is normalized, and is backmapped into the WC system or one isometric to it, and a new intensity calculation is performed using any illumination model. Figure 16.20 shows two edge normals and the normals interpolated from them, before and after normalization.

Color Plates II.31 and II.32 were generated using Gouraud shading and Phong shading respectively, and an illumination equation with a specular-reflectance term. Phong shading yields substantial improvements over Gouraud shading when such illumination models are used, because highlights are reproduced more faithfully, as shown in Fig. 16.21. Consider what happens if n in the Phong $\cos^n \alpha$ illumination term is large and one vertex has a very small α, but each of its adjacent vertices has a large α. The intensity associated with the vertex that has a small α will be appropriate for a highlight, whereas the other vertices will have nonhighlight intensities. If Gouraud shading is used, then the intensity across the polygon is linearly interpolated between the highlight intensity and the lower intensities of the adjacent vertices, spreading the highlight over the polygon (Fig. 16.21a). Contrast this with the sharp drop from the highlight intensity that is computed if linearly interpolated normals are used to compute the $\cos^n \alpha$ term at each pixel (Fig. 16.21b). Furthermore, if a highlight fails to fall at a vertex, then Gouraud shading may miss it entirely (Fig. 16.21c), since no interior point can be brighter than the brightest vertex from which it is interpolated. In contrast, Phong shading allows highlights to be located in a polygon's interior (Fig. 16.21d). Compare the highlights on the ball in Color Plates II.31 and II.32.

Even with an illumination model that does not take into account specular reflectance, the results of normal-vector interpolation are in general superior to intensity interpolation, because an approximation to the normal is used at each point. This reduces Mach-band problems in most cases, but greatly increases the cost of shading in a straightforward implementation, since the interpolated normal must be normalized every time it is used in an illumination model. Duff [DUFF79] has developed a combination of difference

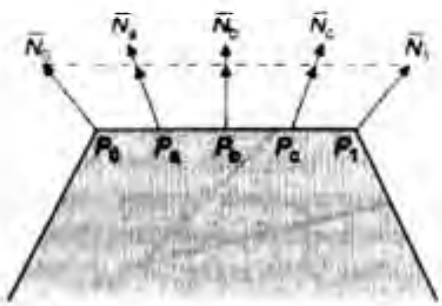


Fig. 16.20 Normal vector interpolation. (After [BUIT75].)

(a) (b) (c) (d)

Fig. 16.21 A specular-reflection illumination model used with Gouraud shading and Phong shading. Highlight falls at left vertex: (a) Gouraud shading, (b) Phong shading. Highlight falls in polygon interior: (c) Gouraud shading, (d) Phong shading. (By David Kurlander, Columbia University.)

equations and table lookup to speed up the calculation. Bishop and Weimer [BISH86] provide an excellent approximation of Phong shading by using a Taylor series expansion that offers even greater increases in shading speed.

Another shading model, intermediate in complexity between Gouraud and Phong shading, involves the linear interpolation of the dot products used in the illumination models. As in Phong shading, the illumination model is evaluated at each pixel, but the interpolated dot products are used to avoid the expense of computing and normalizing any of the direction vectors. This model can produce more satisfactory effects than Gouraud shading when used with specular-reflection illumination models, since the specular term is calculated separately and has power-law, rather than linear, falloff. As in Gouraud shading, however, highlights are missed if they do not fall at a vertex, since no intensity value computed for a set of interpolated dot products can exceed those computed for the set of dot products at either end of the span.

16.2.6 Problems with Interpolated Shading

There are many problems common to all these interpolated-shading models, several of which we list here.

Polygonal silhouette. No matter how good an approximation an interpolated shading model offers to the actual shading of a curved surface, the silhouette edge of the mesh is still clearly polygonal. We can improve this situation by breaking the surface into a greater number of smaller polygons, but at a corresponding increase in expense.

Perspective distortion. Anomalies are introduced because interpolation is performed after perspective transformation in the 3D screen-coordinate system, rather than in the WC system. For example, linear interpolation causes the shading information in Fig. 16.19 to be incremented by a constant amount from one scan line to another along each edge. Consider what happens when vertex 1 is more distant than vertex 2. Perspective foreshortening means that the difference from one scan line to another in the untransformed z value along an edge increases in the direction of the farther coordinate. Thus, if $y_s = (y_1 + y_2) / 2$, then $I_s = (I_1 + I_2) / 2$, but z_s will not equal $(z_1 + z_2) / 2$. This problem can also be

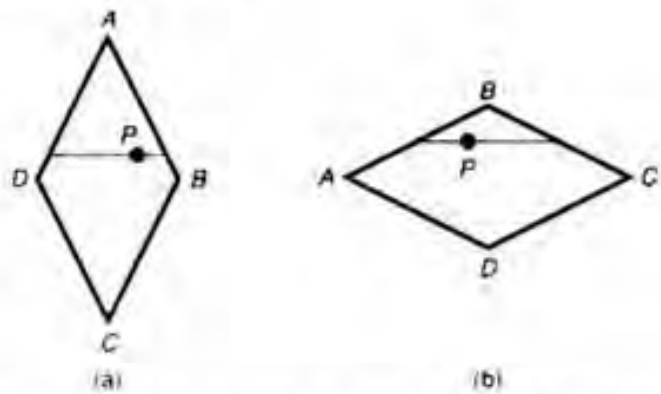


Fig. 16.22 Interpolated values derived for point *P* on the same polygon at different orientations differ from (a) to (b). *P* interpolates *A, B, D* in (a) and *A, B, C* in (b).

reduced by using a larger number of smaller polygons. Decreasing the size of the polygons increases the number of points at which the information to be interpolated is sampled, and therefore increases the accuracy of the shading.

Orientation dependence. The results of interpolated-shading models are not independent of the projected polygon's orientation. Since values are interpolated between vertices and across horizontal scan lines, the results may differ when the polygon is rotated (see Fig. 16.22). This effect is particularly obvious when the orientation changes slowly between successive frames of an animation. A similar problem can also occur in visible-surface determination when the z value at each point is interpolated from the z values assigned to each vertex. Both problems can be solved by decomposing polygons into triangles (see Exercise 16.2). Alternatively, Duff [DUFF79] suggests rotation-independent, but expensive, interpolation methods that solve this problem without the need for decomposition.

Problems at shared vertices. Shading discontinuities can occur when two adjacent polygons fail to share a vertex that lies along their common edge. Consider the three polygons of Fig. 16.23, in which vertex *C* is shared by the two polygons on the right, but not by the large polygon on the left. The shading information determined directly at *C* for the polygons at the right will typically not be the same as the information interpolated at that point from the values at *A* and *B* for the polygon at the left. As a result, there will be a discontinuity in the shading. The discontinuity can be eliminated by inserting in the

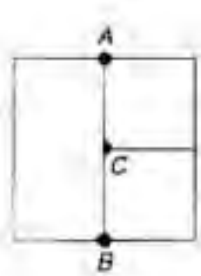


Fig. 16.23 Vertex *C* is shared by the two polygons on the right, but not by the larger rectangular polygon on the left.

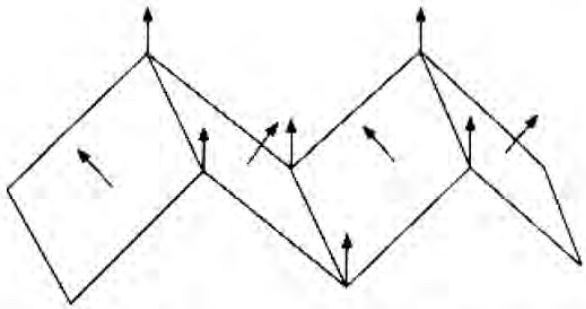


Fig. 16.24 Problems with computing vertex normals. Vertex normals are all parallel.

polygon on the left an extra vertex that shares C's shading information. We can preprocess a static polygonal database in order to eliminate this problem; alternatively, if polygons will be split on the fly (e.g., using the BSP-tree visible-surface algorithm), then extra bookkeeping can be done to introduce a new vertex in an edge that shares an edge that is split.

Unrepresentative vertex normals. Computed vertex normals may not adequately represent the surface's geometry. For example, if we compute vertex normals by averaging the normals of the surfaces sharing a vertex, all of the vertex normals of Fig. 16.24 will be parallel to one another, resulting in little or no variation in shade if the light source is distant. Subdividing the polygons further before vertex normal computation will solve this problem.

Although these problems have prompted much work on rendering algorithms that handle curved surfaces directly, polygons are sufficiently faster (and easier) to process that they still form the core of most rendering systems.

16.3 SURFACE DETAIL

Applying any of the shading models we have described so far to planar or bicubic surfaces produces smooth, uniform surfaces—in marked contrast to most of the surfaces we see and feel. We discuss next a variety of methods developed to simulate this missing surface detail.

16.3.1 Surface-Detail Polygons

The simplest approach adds gross detail through the use of *surface-detail polygons* to show features (such as doors, windows, and lettering) on a base polygon (such as the side of a building). Each surface-detail polygon is coplanar with its base polygon, and is flagged so that it does not need to be compared with other polygons during visible-surface determination. When the base polygon is shaded, its surface-detail polygons and their material properties take precedence for those parts of the base polygon that they cover.

16.3.2 Texture Mapping

As detail becomes finer and more intricate, explicit modeling with polygons or other geometric primitives becomes less practical. An alternative is to map an image, either

digitized or synthesized, onto a surface, a technique pioneered by Catmull [CATM74b] and refined by Blinn and Newell [BLIN76]. This approach is known as *texture mapping* or *pattern mapping*; the image is called a *texture map*, and its individual elements are often called *texels*. The rectangular texture map resides in its own (u, v) texture coordinate space. Alternatively, the texture may be defined by a procedure. Color Plate II.35 shows several examples of texture mapping, using the textures shown in Fig. 16.25. At each rendered pixel, selected texels are used either to substitute for or to scale one or more of the surface's material properties, such as its diffuse color components. One pixel is often covered by a number of texels. To avoid aliasing problems, we must consider all relevant texels.

As shown in Fig. 16.26, texture mapping can be accomplished in two steps. A simple approach starts by mapping the four corners of the pixel onto the surface. For a bicubic patch, this mapping naturally defines a set of points in the surface's (s, t) coordinate space. Next, the pixel's corner points in the surface's (s, t) coordinate space are mapped into the texture's (u, v) coordinate space. The four (u, v) points in the texture map define a quadrilateral that approximates the more complex shape into which the pixel may actually map due to surface curvature. We compute a value for the pixel by summing all texels that lie within the quadrilateral, weighting each by the fraction of the texel that lies within the quadrilateral. If a transformed point in (u, v) space falls outside of the texture map, the texture map may be thought of as being replicated, like the patterns of Section 2.1.3.

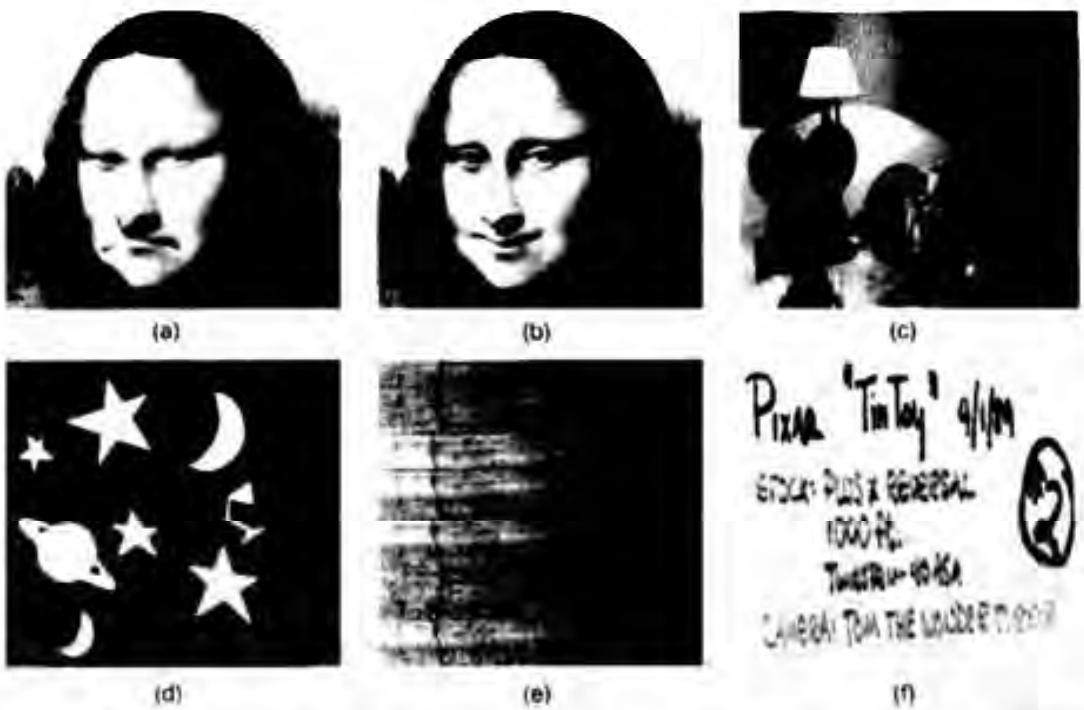


Fig. 16.25 Textures used to create Color Plate II.35. (a) Frowning Mona. (b) Smiling Mona. (c) Painting. (d) Wizard's cap. (e) Floor. (f) Film label. (Copyright © 1990, Pixar. Images rendered by Thomas Williams and H. B. Siegel using Pixar's PhotoRealistic RenderMan™ software.)

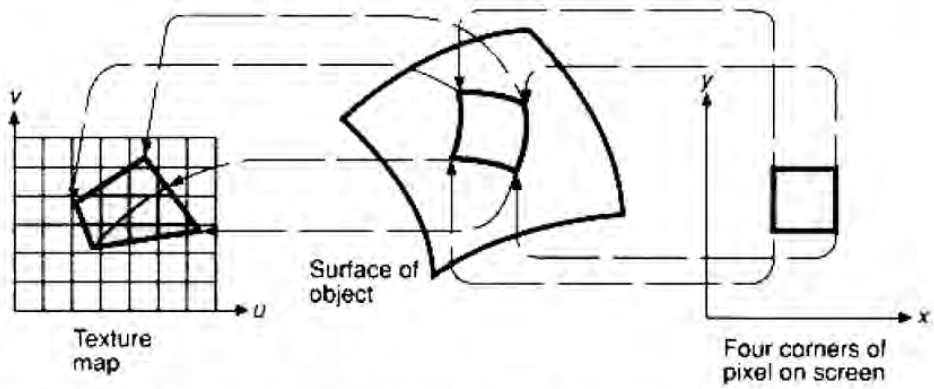


Fig. 16.26 Texture mapping from pixel to the surface to the texture map.

Rather than always use the identity mapping between (s, t) and (u, v), we can define a correspondence between the four corners of the 0-to-1 (s, t) rectangle and a quadrilateral in (u, v). When the surface is a polygon, it is common to assign texture map coordinates directly to its vertices. Since, as we have seen, linearly interpolating values across arbitrary polygons is orientation-dependent, polygons may be decomposed into triangles first. Even after triangulation, however, linear interpolation will cause distortion in the case of perspective projection. This distortion will be more noticeable than that caused when interpolating other shading information, since texture features will not be correctly foreshortened. We can obtain an approximate solution to this problem by decomposing polygons into smaller ones, or an exact solution, at greater cost, by performing the perspective division while interpolating.

The approach just described assumes square pixel geometry and simple box filtering. It also fails to take into account pixels that map to only part of a surface. Feibush, Levoy, and Cook [FEIB80] address these problems for texture-mapping polygons. Think of the square pixel in Fig. 16.26 as the bounding rectangle of the support of an arbitrary filter centered at a pixel. Pixels and texels can then be treated as points. In effect, all texels that lie within the mapped intersection of the transformed bounding rectangle and polygon are selected, and these texels' coordinates are transformed into the coordinate system of the bounding rectangle. Each texel's transformed coordinates are used to index into a filter table to determine the texel's weighting, and the weighted average of the texel intensities is computed. This weighted average must in turn be weighted by the percentage contribution that the polygon makes to the pixel's intensity. The process is repeated for each polygon whose projection intersects the pixel, and the values are summed. Section 17.4.2 discusses this algorithm in more detail.

The Feibush, Levoy, and Cook algorithm can be quite inefficient. Consider mapping a checkerboard pattern onto an infinite ground plane. An extremely large number of texels may have to be weighted and summed just to texture a single distant ground-plane pixel. One solution to this problem is to prefilter the texture and to store the results in a way that is space-efficient and that allows quick determination of the weighted average of texels mapping to a pixel. Algorithms by Williams [WILL83], Crow [CROW84], Glassner [GLAS86], and Heckbert [HECK86a] that take this approach are discussed in Section

17.4.3. Catmull and Smith's efficient technique [CATM80] for mapping an entire texture map directly to a surface is discussed in Exercise 17.10. Heckbert [HECK86] provides a thorough survey of texture-mapping methods.

16.3.3 Bump Mapping

Texture mapping affects a surface's shading, but the surface continues to appear geometrically smooth. If the texture map is a photograph of a rough surface, the surface being shaded will not look quite right, because the direction to the light source used to create the texture map is typically different from the direction to the light source illuminating the surface. Blinn [BLIN78b] developed a way to provide the appearance of modified surface geometry that avoids explicit geometrical modeling. His approach involves perturbing the surface normal before it is used in the illumination model, just as slight roughness in a surface would perturb the surface normal. This method is known as *bump mapping*, and is based on texture mapping.

A *bump map* is an array of displacements, each of which can be used to simulate displacing a point on a surface a little above or below that point's actual position. Let us represent a point on a surface by a vector $\bar{P}$, where $\bar{P} = [x(s, t), y(s, t), z(s, t)]$. We call the partial derivatives of the surface at $\bar{P}$ with respect to the surface's s and t parameterization axes, $\bar{P}_s$ and $\bar{P}_t$. Since each is tangent to the surface, their cross-product forms the (unnormalized) surface normal at $\bar{P}$. Thus,

$$\bar{N} = \bar{P}_s \times \bar{P}_t. \tag{16.21}$$

We can displace point $\bar{P}$ by adding to it the normalized normal scaled by a selected bump-map value B. The new point is

$$\bar{P}' = \bar{P} + \frac{B\bar{N}}{|\bar{N}|}. \tag{16.22}$$

Blinn shows that a good approximation to the new (unnormalized) normal $\bar{N}'$ is

$$\bar{N}' = \bar{N} + \frac{B_u(\bar{N} \times \bar{P}_t) - B_v(\bar{N} \times \bar{P}_s)}{|\bar{N}|}, \tag{16.23}$$

where B_u and B_v are the partial derivatives of the selected bump-map entry B with respect to the bump-map parameterization axes, u and v. $\bar{N}'$ is then normalized and substituted for the surface normal in the illumination equation. Note that only the partial derivatives of the bump map are used in Eq. (16.23), not its values. Bilinear interpolation can be used to derive bump-map values for specified (u, v) positions, and finite differences can be used to compute B_u and B_v.

The results of bump mapping can be quite convincing. Viewers often fail to notice that an object's texture does not affect its silhouette edges. Color Plates III.3 and III.4 show two examples of bump mapping. Unlike texture mapping, aliasing cannot be dealt with by filtering values from the bump map, since these values do not correspond linearly to intensities; filtering the bump map just smooths out the bumps. Instead, subpixel intensities may be computed and filtered for each pixel, or some prefiltering may be performed on the bump map to improve gross aliasing.

16.3.4 Other Approaches

Although 2D mapping can be effective in many situations, it often fails to produce convincing results. Textures frequently betray their 2D origins when mapped onto curved surfaces, and problems are encountered at texture "seams." For example, when a wood-grain texture is mapped onto the surface of a curved object, the object will look as if it were painted with the texture. Peachey [PEAC85] and Perlin [PERL85] have investigated the use of solid textures for proper rendering of objects "carved" of wood or marble, as exemplified by Color Plate IV.21. In this approach, described in Section 20.8.3, the texture is a 3D function of its position in the object.

Other surface properties can be mapped as well. For example, Gardner [GARD84] has used transparency mapping to make impressionistic trees and clouds from otherwise simple shapes, as described in Section 20.8.2. Color Plate IV.24 shows the application of a complex functional transparency texture to objects formed from groups of quadric surfaces. Cook has implemented *displacement mapping*, in which the actual surface is displaced, instead of only the surface normals [COOK84a]; this process, which must be carried out before visible-surface determination, was used to modify the surfaces of the cone and torus in Color Plate II.36. Using fractals to create richly detailed geometry from an initial simple geometric description is discussed in Section 20.3.

So far, we have made the tacit assumption that the process of shading a point on an object is unaffected by the rest of that object or by any other object. But an object might in fact be shadowed by another object between it and a light source; might transmit light, allowing another object to be seen through it; or might reflect other objects, allowing another object to be seen because of it. In the following sections, we describe how to model these effects.

16.4 SHADOWS

Visible-surface algorithms determine which surfaces can be seen from the viewpoint; shadow algorithms determine which surfaces can be "seen" from the light source. Thus, visible-surface algorithms and shadow algorithms are essentially the same. The surfaces that are visible from the light source are not in shadow; those that are not visible from the light source are in shadow. When there are multiple light sources, a surface must be classified relative to each of them.

Here, we consider shadow algorithms for point light sources; extended light sources are discussed in Sections 16.8, 16.12, and 16.13. Visibility from a point light source is, like visibility from the viewpoint, all or nothing. When a point on a surface cannot be seen from a light source, then the illumination calculation must be adjusted to take it into account. The addition of shadows to the illumination equation yields

$$I_\lambda = I_{a\lambda} k_a O_{d\lambda} + \sum_{1 \le i \le m} S_i f_{att_i} I_{p\lambda_i} [k_d O_{d\lambda} (\overline{N} \cdot \overline{L}_i) + k_s O_{s\lambda} (\overline{R}_i \cdot \overline{V})^n], \qquad (16.24)$$

where

$$S_i = \begin{cases} 0, \text{ if light } i \text{ is blocked at this point;} \\ 1, \text{ if light } i \text{ is not blocked at this point.} \end{cases}$$

Note that areas in the shadow of all point light sources are still illuminated by the ambient light.

Although computing shadows requires computing visibility from the light source, as we have pointed out, it is also possible to generate "fake" shadows without performing any visibility tests. These can be created efficiently by transforming each object into its polygonal projection from a point light source onto a designated ground plane, without clipping the transformed polygon to the surface that it shadows or checking for whether it is blocked by intervening surfaces [BLIN88]. These shadows are then treated as surface-detail polygons. For the general case, in which these fake shadows are not adequate, various approaches to shadow generation are possible. We could perform all shadow processing first, interleave it with visible-surface processing in a variety of ways, or even do it after visible-surface processing has been performed. Here we examine algorithms that follow each of these approaches, building on the classification of shadow algorithms presented in [CROW77a]. To simplify the explanations, we shall assume that all objects are polygons unless otherwise specified.

16.4.1 Scan-Line Generation of Shadows

One of the oldest methods for generating shadows is to augment a scan-line algorithm to interleave shadow and visible-surface processing [APPEL68; BOUK70b]. Using the light source as a center of projection, the edges of polygons that might potentially cast shadows are projected onto the polygons intersecting the current scan line. When the scan crosses one of these shadow edges, the colors of the image pixels are modified accordingly.

A brute-force implementation of this algorithm must compute all $n(n-1)$ projections of every polygon on every other polygon. Bouknight and Kelley [BOUK70b] instead use a clever preprocessing step in which all polygons are projected onto a sphere surrounding the light source, with the light source as center of projection. Pairs of projections whose extents do not overlap can be eliminated, and a number of other special cases can be identified to limit the number of polygon pairs that need be considered by the rest of the algorithm. The authors then compute the projection from the light source of each polygon onto the plane of each of those polygons that they have determined it could shadow, as shown in Fig. 16.27. Each of these shadowing polygon projections has associated information about the polygons casting and potentially receiving the shadow. While the scan-line algorithm's regular scan keeps track of which regular polygon edges are being crossed, a separate, parallel shadow scan keeps track of which shadowing polygon projection edges are crossed, and thus which shadowing polygon projections the shadow scan is currently "in." When the shade for a span is computed, it is in shadow if the shadow scan is "in" one of the shadow projections cast on the polygon's plane. Thus span *bc* in Fig. 16.27(a) is in shadow, while spans *ab* and *cd* are not. Note that the algorithm does not need to clip the shadowing polygon projections analytically to the polygons being shadowed.

16.4.2 A Two-Pass Object-Precision Shadow Algorithm

Atherton, Weiler, and Greenberg have developed an algorithm that performs shadow determination before visible-surface determination [ATHE78]. They process the object description by using the same algorithm twice, once for the viewpoint, and once for the

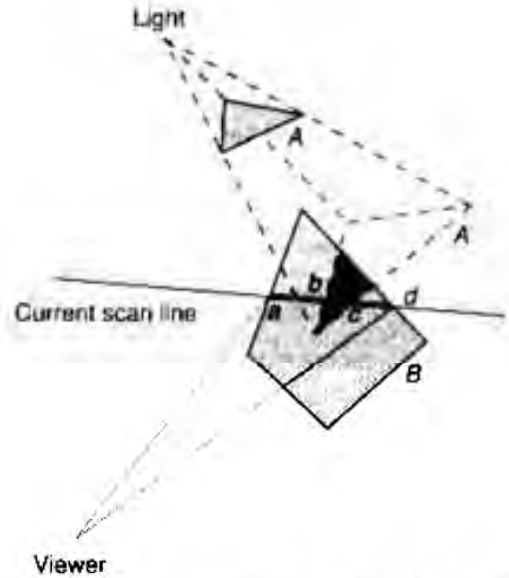
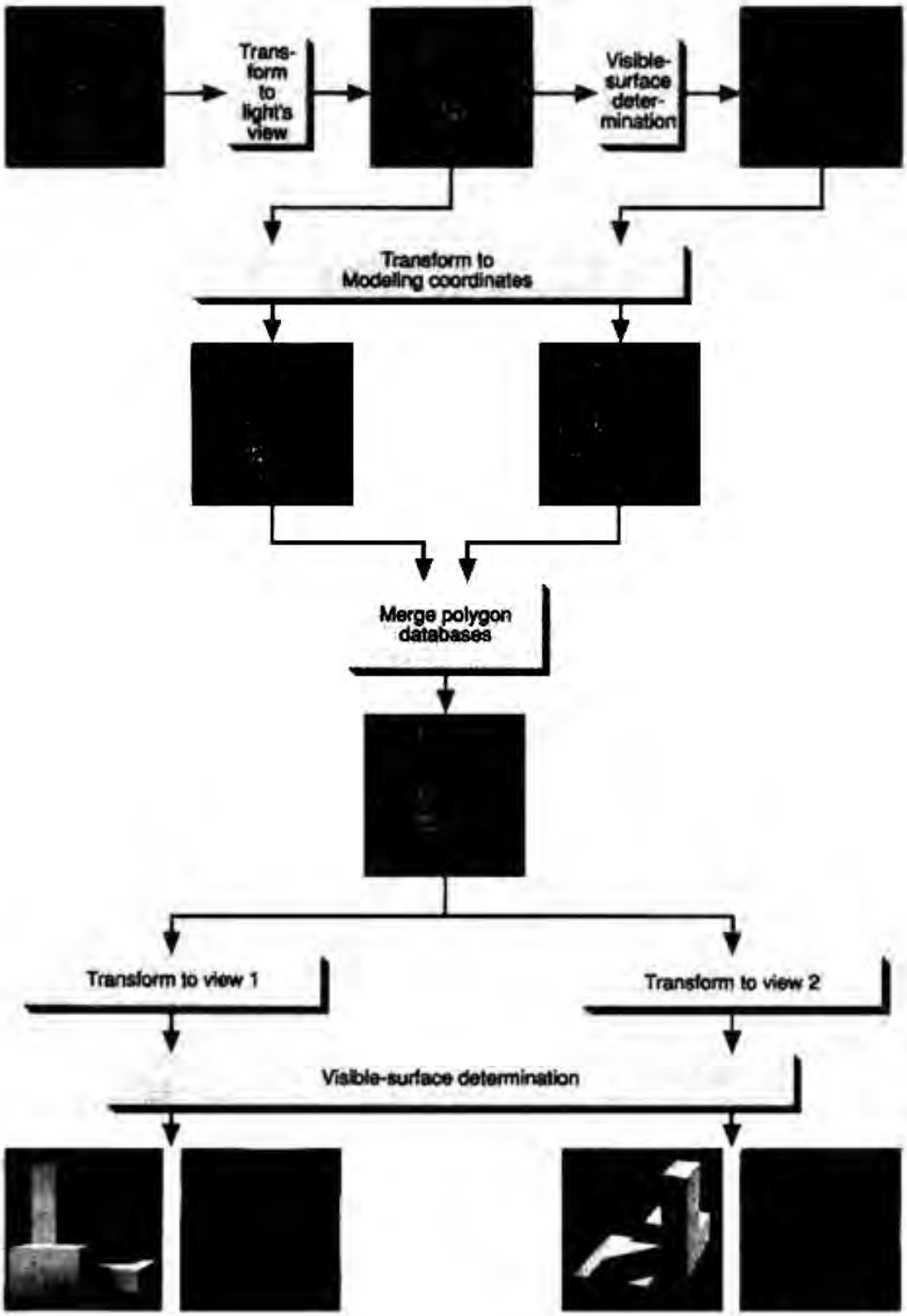


Fig. 16.27 A scan-line shadow algorithm using the Bouknight and Kelley approach. Polygon *A* casts shadow *A'* on plane of *B*.

light source. The results are then combined to determine the pieces of each visible part of a polygon that are lit by the light source, and the scene is scan-converted. Thus, since the shadows are not dependent on the viewpoint, all the shadow calculations may be performed just once for a series of images of the same objects seen from many different viewpoints, as long as the light source and objects are fixed.

The algorithm, shown in overview in Fig. 16.28, first determines those surfaces that are visible from the light source's viewpoint, using the Weiler–Atherton visible-surface algorithm discussed in Section 15.7.2. The output of this pass is a list of lit polygons, each of which is tagged with the identity of its parent polygon. All the objects must fit into the light source's view volume, since parts of the objects that do not fit are not recognized as lit. If a light source's view volume cannot encompass all the objects, multiple nonoverlapping view volumes can be constructed that radiate out from the light source, a technique called *sectoring*.

Next, the lit polygons are transformed back into the modeling coordinates and are merged with a copy of the original database as surface-detail polygons (Section 16.3), creating a viewpoint-independent merged database, shown in Fig. 16.29. Note that the implementation illustrated in Fig. 16.28 performs the same transformations on both databases before merging them. Hidden-surface removal is then performed on a copy of this merged database from the viewpoint of an arbitrary observer, again using the Weiler–Atherton algorithm. All processing so far is performed with object precision and results in a list of polygons. A polygon scan-conversion algorithm is then used to render the image. Visible surfaces covered by surface-detail polygons are rendered as lit, whereas uncovered visible surfaces are rendered in shadow. Color Plate III.5 was generated using this approach. Multiple light sources can be handled by processing the merged database from the viewpoint of each new light source, merging the results of each pass.


Trans-
form
to
light's
view
Visible-
surface
deter-
mination
Transform to
Modeling coordinates
Merge polygon
databases
Transform to view 1
Transform to view 2
Visible-surface determination

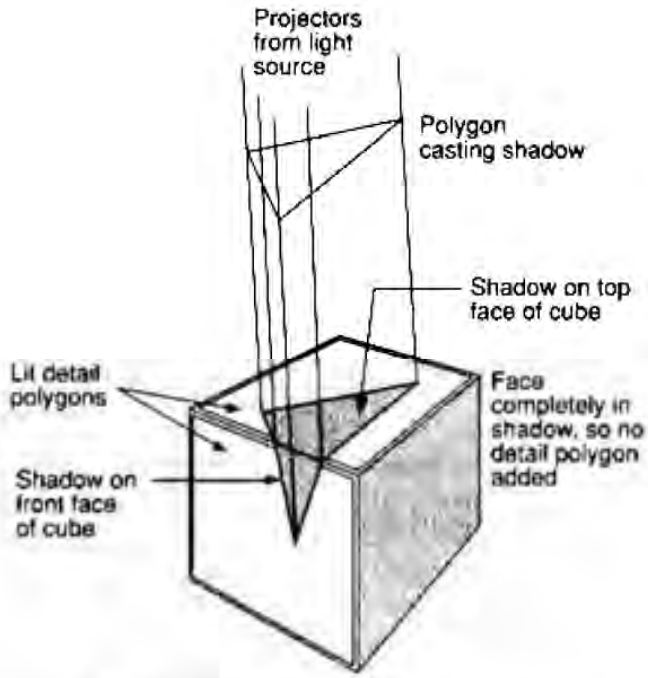


Fig. 16.29 Lit surface-detail polygons.

16.4.3 Shadow Volumes

Crow [CROW77a] describes how to generate shadows by creating for each object a *shadow volume* that the object blocks from the light source. A shadow volume is defined by the light source and an object and is bounded by a set of invisible *shadow polygons*. As shown in Fig. 16.30, there is one quadrilateral shadow polygon for each silhouette edge of the object relative to the light source. Three sides of a shadow polygon are defined by a silhouette edge of the object and the two lines emanating from the light source and passing through that edge's endpoints. Each shadow polygon has a normal that points out of the shadow volume. Shadow volumes are generated only for polygons facing the light. In the implementation described by Bergeron [BERG86a], the shadow volume—and hence each of its shadow polygons—is capped on one end by the original object polygon and on the other end by a scaled copy of the object polygon whose normal has been inverted. This scaled copy is located at a distance from the light beyond which its attenuated energy density is assumed to be negligible. We can think of this distance as the light's *sphere of influence*. Any point outside of the sphere of influence is effectively in shadow and does not require any additional shadow processing. In fact, there is no need to generate a shadow volume for any object wholly outside the sphere of influence. We can generalize this approach to apply to nonuniformly radiating sources by considering a *region of influence*, for example by culling

Fig. 16.28 Shadow creation and display in the Atherton, Weiler, and Greenberg algorithm. (Images by Peter Atherton, Kevin Weiler, Donald P. Greenberg, Program of Computer Graphics, Cornell University, 1978.)

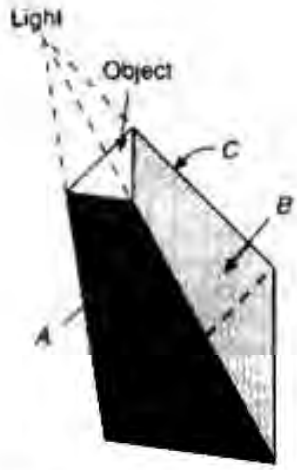


Fig. 16.30 A shadow volume is defined by a light source and an object.

objects outside of a light's flaps and cone. The shadow volume may also be further clipped to the view volume if the view volume is known in advance. The cap polygons are also treated as shadow polygons by the algorithm.

Shadow polygons are not rendered themselves, but are used to determine whether the other objects are in shadow. Relative to the observer, a front-facing shadow polygon (polygon *A* or *B* in Fig. 16.30) causes those objects behind it to be shadowed; a back-facing shadow polygon (polygon *C*) cancels the effect of a front-facing one. Consider a vector from the viewpoint *V* to a point on an object. The point is in shadow if the vector intersects more front-facing than back-facing shadow polygons. Thus, points *A* and *C* in Fig. 16.31(a) are in shadow. This is the only case in which a point is shadowed when *V* is not shadowed; therefore, point *B* is lit. If *V* is in shadow, there is one additional case in which a point is shadowed: when all the back-facing shadow polygons for the object polygons shadowing the eye have not yet been encountered. Thus, points *A*, *B*, and *C* in Fig. 16.31(b) are in shadow.

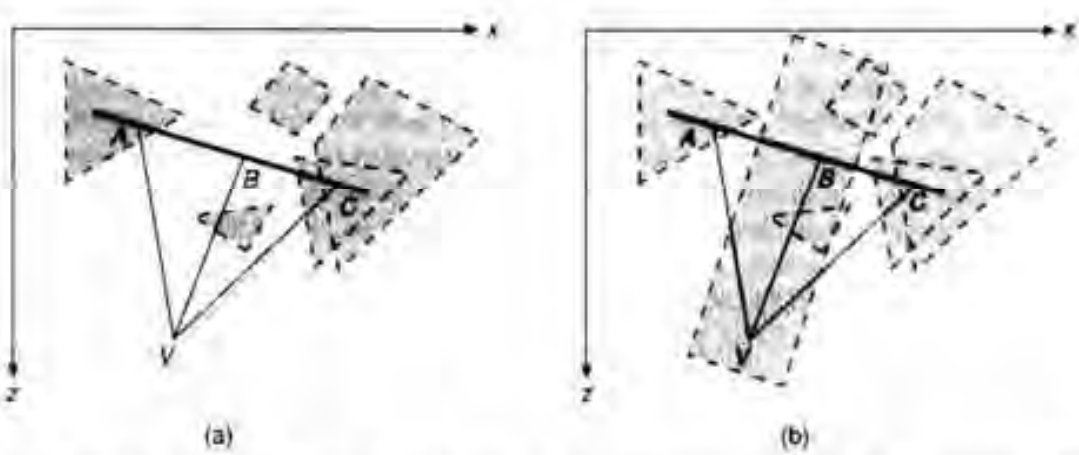


Fig. 16.31 Determining whether a point is in shadow for a viewer at *V*. Dashed lines define shadow volumes (shaded in gray). (a) *V* is not in shadow. Points *A* and *C* are shadowed, point *B* is lit. (b) *V* is in shadow. Points *A*, *B*, and *C* are shadowed.

even though the vector from V to B intersects the same number of front-facing and back-facing shadow polygons as it does in part (a).

We can compute whether a point is in shadow by assigning to each front-facing (relative to the viewer) shadow polygon a value of $+1$ and to each back-facing shadow polygon a value of -1. A counter is initially set to the number of shadow volumes that contain the eye and is incremented by the values associated with all shadow polygons between the eye and the point on the object. The point is in shadow if the counter is positive at the point. The number of shadow volumes containing the eye is computed only once for each viewpoint, by taking the negative of the sum of the values of all shadow polygons intercepted by an arbitrary projector from the eye to infinity.

Although is is possible to compute a shadow volume for each polygon, we can take advantage of object coherence by computing a single shadow volume for each connected polyhedron. This can be accomplished by generating shadow polygons from only those edges that are silhouette edges relative to the light source; these are the contour edges relative to the light source (as defined in Section 15.3.2).

Multiple light sources can be handled by building a separate set of shadow volumes for each light source, marking the volume's shadow polygons with their light source identifier, and keeping a separate counter for each light source. Brotman and Badler [BROT84] have implemented a z-buffer version of the shadow-volume algorithm, and Bergeron [BERG86a] discusses a scan-line implementation that efficiently handles arbitrary polyhedral objects containing nonplanar polygons.

Chin and Feiner [CHIN89] describe an object-precision algorithm that builds a single shadow volume for a polygonal environment, using the BSP-tree solid modeling representation discussed in Section 12.6.4. Polygons are processed in front-to-back order relative to the light source. Each polygon facing the light source is filtered down the tree, dividing the polygon into lit and shadowed fragments. Only lit fragments cast shadows, so the semi-infinite pyramid defined by the light source and each lit fragment is added to the volume. Because of the front-to-back order, every polygon is guaranteed not to lie between the light source and the polygons processed previously. Therefore, since no polygon needs to be compared with the plane of a previously processed polygon, the polygons themselves do not need to be added to the shadow volume. As with the Atherton–Weiler–Greenberg algorithm, the lit fragments may be added to the environment as surface-detail polygons or the lit and shadowed fragments may be displayed together instead. Multiple light sources are accommodated by filtering the polygon fragments of one shadow-volume BSP tree down the shadow-volume BSP tree of the next light source. Each fragment is tagged to indicate the light sources that illuminate it, allowing the resulting fragmented environment to be displayed with any polygon visible-surface algorithm. Because of the shadow volume representation, lights may be positioned anywhere relative to the objects; thus, sectoring is not necessary. Several optimizations and a parallel version of the algorithm are discussed in [CHIN90]. Color Plate III.6(a) is rendered with the algorithm; Color Plate III.6(b) shows the fragments created in filtering the polygons down the shadow-volume BSP tree.

16.4.4 A Two-Pass z-Buffer Shadow Algorithm

Williams [WILL78] developed a shadow-generation method based on two passes through a z-buffer algorithm, one for the viewer and one for the light source. His algorithm, unlike the two-pass algorithm of Section 16.4.2, determines whether a surface is shadowed by using

image-precision calculations. Figure 16.32(a) shows an overview of an environment lit by a light at L; a shadowless image from viewpoint V is shown in Fig. 16.32(d). The algorithm begins by calculating and storing just the z-buffer for the image from the viewpoint of the light (Fig. 16.32b). In Fig. 16.32(b), increasing intensities represent increasing distance. Next, the z-buffer (Fig. 16.32c) and the image (Fig. 16.32e) are calculated from the viewpoint of the observer using a z-buffer algorithm with the following modification. Whenever a pixel is determined to be visible, its object-precision coordinates in the observer's view (x_0, y_0, z_0) are transformed into coordinates in the light source's view (x'_0, y'_0, z'_0). The transformed coordinates x'_0 and y'_0 are used to select the value z_L in the light source's z-buffer to be compared with the transformed value z'_0. If z_L is closer to the light than is z'_0, then there is something blocking the light from the point, and the pixel is shaded as being in shadow; otherwise the point is visible from the light and it is shaded as lit. In analogy to texture mapping, we can think of the light's z-buffer as a *shadow map*. Multiple light sources can be accommodated by use of a separate shadow map for each light source.

Like the regular z-buffer visible-surface algorithm, this algorithm requires that each rendered pixel be shaded. Here, this means that shadow calculations must be performed for the pixel, even if it is ultimately painted over by closer objects. Williams has suggested a variation on his algorithm that exploits the ease with which the z-buffer algorithm can interleave visible-surface determination with illumination and shading, and eliminates shadow calculations for obscured objects. Rather than computing just the shadow map first, the modified algorithm also computes the regular shaded image from the observer's point of view (Fig. 16.32d), along with its z-buffer (all these computations can use conventional z-buffer–based hardware). Shadows are then added using a postprocess that is linear in the

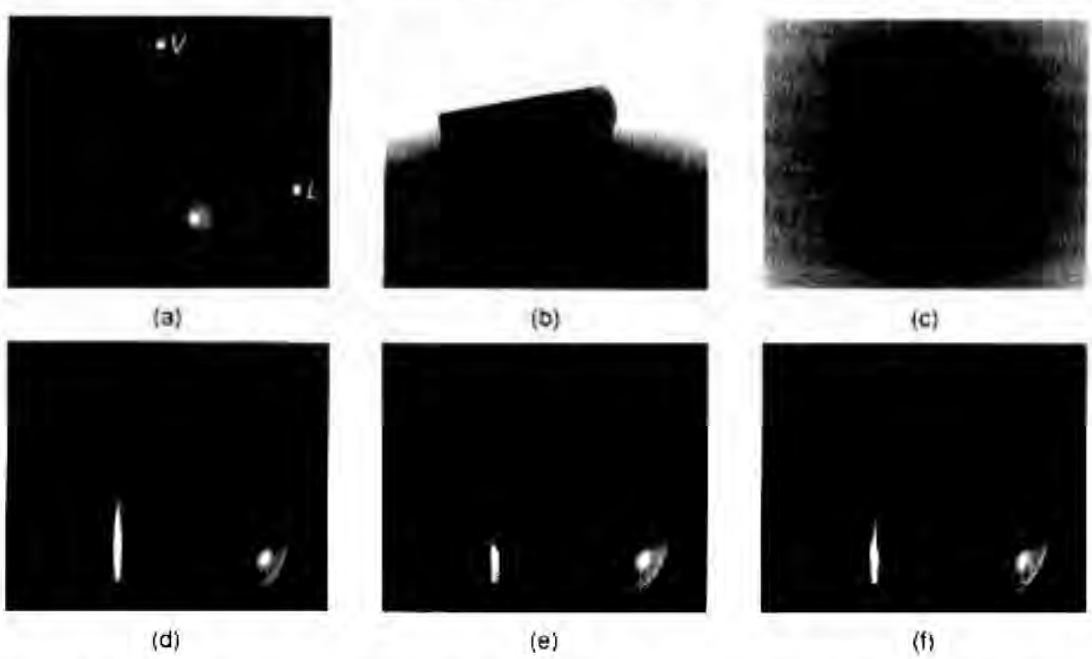


Fig. 16.32 z-buffer shadow-generation method. (a) Overview. (b) Light's z-buffer. (c) Observer's z-buffer. (d) Observer's image. (e) Observer's image with shadows. (f) Observer's image with post-processed shadows. (By David Kurlander, Columbia University.)

Fig. 16.33 Shadow map used to create Color Plate II.36. (Copyright © 1990, Pixar. Shadow map rendered by Thomas Williams and H. B. Siegel using Pixar's PhotoRealistic RenderMan™ software.)

number of pixels in the image to produce Fig. 16.32(f). The same transformation and comparison operation as before are performed for each pixel in the observer's image. If z_L is closer to the light than is z'_u, then the pixel's previously computed shade in the observer's image is darkened. Although this approach is significantly more efficient than is the original algorithm, it results in artifacts; most noticeably, shadowed objects will have (darkened) specular highlights, even though there should be no specular highlights on an object that is shielded from the light source. In addition, the z_u to be transformed is at object precision in the first version, but at the typically lower z-buffer precision here. (See Exercise 16.15.)

Unlike the other shadow algorithms discussed so far, Williams's algorithm makes it especially easy to generate shadows for any object that can be scan-converted, including curved surfaces. Because all operations are performed in image precision, however, allowance must be made for the limited numerical precision. For example, the transformation from z_u to z'_u should also move z'_u a little closer to the light source, to avoid having a visible point cast a shadow on itself. Like the z-buffer visible-surface algorithm from which it is constructed, this shadow algorithm is prone to aliasing. Williams describes how filtering and dithering can reduce the effects of aliasing. Reeves, Salesin, and Cook [REEV87] demonstrate improvements using *percentage closer filtering*. Each z'_u is compared with values in a region of the shadow map, and the percentage of closer values determines the amount of shadowing. This improved algorithm was used to render Color Plates D, F, and II.36. Figure 16.33 shows the shadow map used to create Color Plate II.36.

16.4.5 Global Illumination Shadow Algorithms

Ray-tracing and radiosity algorithms have been used to generate some of the most impressive pictures of shadows in complex environments. Simple ray tracing has been used to model shadows from point light sources, whereas more advanced versions allow extended light sources. Both are discussed in Section 16.12. Radiosity methods, discussed in Section 16.13, model light sources as light-emitting surfaces that may have the same geometry as any other surface; thus, they implicitly support extended light sources.

16.5 TRANSPARENCY

Much as surfaces can have specular and diffuse reflection, those that transmit light can be transparent or translucent. We can usually see clearly through *transparent* materials, such as glass, although in general the rays are refracted (bent). Diffuse transmission occurs through *translucent* materials, such as frosted glass. Rays passing through translucent materials are jumbled by surface or internal irregularities, and thus objects seen through translucent materials are blurred.

16.5.1 Nonrefractive Transparency

The simplest approach to modeling transparency ignores refraction, so light rays are not bent as they pass through the surface. Thus, whatever is visible on the line of sight through a transparent surface is also geometrically located on that line of sight. Although refractionless transparency is not realistic, it can often be a more useful effect than refraction. For example, it can provide a distortionless view through a surface, as depicted in Color Plate III.7. As we have noted before, total photographic realism is not always the objective in making pictures.

Two different methods have been commonly used to approximate the way in which the colors of two objects are combined when one object is seen through the other. We shall refer to these as *interpolated* and *filtered* transparency.

Interpolated transparency. Consider what happens when transparent polygon 1 is between the viewer and opaque polygon 2, as shown in Fig. 16.34. *Interpolated transparency* determines the shade of a pixel in the intersection of two polygons' projections by linearly interpolating the individual shades calculated for the two polygons:

$$I_\lambda = (1 - k_{t_1})I_{\lambda_1} + k_{t_1}I_{\lambda_2}. \tag{16.25}$$

The *transmission coefficient* k_{t_1} measures the *transparency* of polygon 1, and ranges between 0 and 1. When k_{t_1} is 0, the polygon is opaque and transmits no light; when k_{t_1} is 1, the polygon is perfectly transparent and contributes nothing to the intensity I_λ. The value $1 - k_{t_1}$ is called the polygon's *opacity*. Interpolated transparency may be thought of as modeling a polygon that consists of a fine mesh of opaque material through which other objects may be seen; k_{t_1} is the fraction of the mesh's surface that can be seen through. A totally transparent

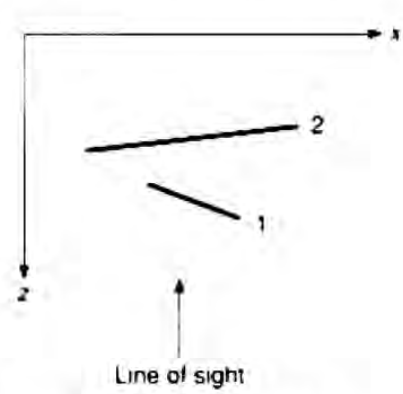


Fig. 16.34 Cross-section of two polygons.

polygon that is processed this way will not have any specular reflection. For a more realistic effect, we can interpolate only the ambient and diffuse components of polygon 1 with the full shade of polygon 2, and then add in polygon 1's specular component [KAY79b].

Another approach, often called *screen-door transparency*, literally implements a mesh by rendering only some of the pixels associated with a transparent object's projection. The low-order bits of a pixel's (x, y) address are used to index into a transparency bit mask. If the indexed bit is 1, then the pixel is written; otherwise, it is suppressed, and the next closest polygon at that pixel is visible. The fewer 1 bits in the mask, the more transparent the object's appearance. This approach relies on having our eyes perform spatial integration to produce interpolated transparency. Note, however, that an object fully obscures any other object drawn with the same transparency mask and that other undesirable interactions between masks are difficult to avoid.

Filtered transparency. *Filtered transparency* treats a polygon as a transparent filter that selectively passes different wavelengths; it can be modeled by

$$I_\lambda = I_{\lambda_1} + k_{t_1} O_{t\lambda} I_{\lambda_2}, \tag{16.26}$$

where $O_{t\lambda}$ is polygon 1's *transparency color*. A colored filter may be modeled by choosing a different value of $O_{t\lambda}$ for each λ (but see Section 16.9). In either interpolated or filtered transparency, if additional transparent polygons are in front of these polygons, then the calculation is invoked recursively for polygons in back-to-front order, each time using the previously computed I_λ as I_{λ_2}.

Implementing transparency. Several visible-surface algorithms can be readily adapted to incorporate transparency, including scan-line and list-priority algorithms. In list-priority algorithms, the color of a pixel about to be covered by a transparent polygon is read back and used in the illumination model while the polygon is being scan-converted.

Most z-buffer-based systems support screen-door transparency because it allows transparent objects to be intermingled with opaque objects and to be drawn in any order. Adding transparency effects that use Eqs. (16.25) or (16.26) to the z-buffer algorithm is more difficult, because polygons are rendered in the order in which they are encountered. Imagine rendering several overlapping transparent polygons, followed by an opaque one. We would like to slip the opaque polygon behind the appropriate transparent ones. Unfortunately, the z-buffer does not store the information needed to determine which transparent polygons are in front of the opaque polygon, or even the polygons' relative order. One simple, although incorrect, approach is to render transparent polygons last, combining their colors with those already in the frame buffer, but not modifying the z-buffer; when two transparent polygons overlap, however, their relative depth is not taken into account.

Mammen [MAMM89] describes how to render transparent objects properly in back-to-front order in a z-buffer-based system through the use of multiple rendering passes and additional memory. First, all the opaque objects are rendered using a conventional z-buffer. Then, transparent objects are processed into a separate set of buffers that contain, for each pixel, a transparency value and a flag bit, in addition to the pixel's color and z value. Each flag bit is initialized to off and each z value is set to the closest possible value. If a transparent object's z value at a pixel is closer than the value in the opaque z-buffer, but is

more distant than that in the transparent z-buffer, then the color, z value, and transparency are saved in the transparent buffers, and the flag bit is set. After all objects have been processed, the transparent object buffers contain information for the most distant transparent object at each pixel whose flag bit is set. Information for flagged pixels is then blended with that in the original frame buffer and z-buffer. A flagged pixel's transparency z-value replaces that in the opaque z-buffer and the flag bit is reset. This process is repeated to render successively closer objects at each pixel. Color Plate III.7 was made using this algorithm.

Kay and Greenberg [KAY79b] have implemented a useful approximation to the increased attenuation that occurs near the silhouette edges of thin curved surfaces, where light passes through more material. They define k_t in terms of a nonlinear function of the z component of the surface normal after perspective transformation,

$$k_t = k_{t_{min}} + (k_{t_{max}} - k_{t_{min}})(1 - (1 - z_N)^m), \tag{16.27}$$

where $k_{t_{min}}$ and $k_{t_{max}}$ are the object's minimum and maximum transparencies, z_N is the z component of the normalized surface normal at the point for which k_t is being computed, and m is a power factor (typically 2 or 3). A higher m models a thinner surface. This new value of k_t may be used as k_{t_1} in either Eq. (16.25) or (16.26).

16.5.2 Refractive Transparency

Refractive transparency is significantly more difficult to model than is nonrefractive transparency, because the geometrical and optical lines of sight are different. If refraction is considered in Fig. 16.35, object A is visible through the transparent object along the line of sight shown; if refraction is ignored, object B is visible. The relationship between the angle of incidence θ_i and the angle of refraction θ_t is given by Snell's law

$$\frac{\sin \theta_i}{\sin \theta_t} = \frac{\eta_{tA}}{\eta_{iA}}. \tag{16.28}$$

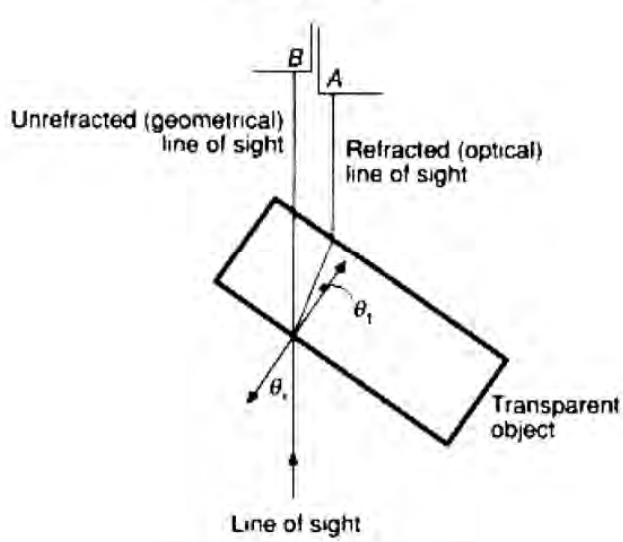


Fig. 16.35 Refraction.

where $\eta_{1\lambda}$ and $\eta_{t\lambda}$ are the *indices of refraction* of the materials through which the light passes. A material's index of refraction is the ratio of the speed of light in a vacuum to the speed of light in the material. It varies with the wavelength of the light and even with temperature. A vacuum has an index of refraction of 1.0, as does the atmosphere to close approximation; all materials have higher values. The index of refraction's wavelength-dependence is evident in many instances of refraction as *dispersion*—the familiar, but difficult to model, phenomenon of refracted light being spread into its spectrum [THOM86; MUSG89].

Calculating the refraction vector. The unit vector in the direction of refraction, $\bar{T}$, can be calculated as

$$\bar{T} = \sin\theta_t\,\bar{M} - \cos\theta_t\,\bar{N}, \tag{16.29}$$

where $\bar{M}$ is a unit vector perpendicular to $\bar{N}$ in the plane of the incident ray $\bar{I}$ and $\bar{N}$ [HECK84] (Fig. 16.36). Recalling the use of $\bar{S}$ in calculating the reflection vector $\bar{R}$ in Section 16.1.4, we see that $\bar{M} = (\bar{N}\cos\theta_i - \bar{I}) / \sin\theta_i$. By substitution,

$$\bar{T} = \frac{\sin\theta_t}{\sin\theta_i}(\bar{N}\cos\theta_i - \bar{I}) - \cos\theta_t\,\bar{N}. \tag{16.30}$$

If we let $\eta_{t\lambda} = \eta_{1\lambda} / \eta_{t\lambda} = \sin\theta_i / \sin\theta_t$, then after rearranging terms

$$\bar{T} = (\eta_{t\lambda}\cos\theta_i - \cos\theta_t)\,\bar{N} - \eta_{t\lambda}\,\bar{I}. \tag{16.31}$$

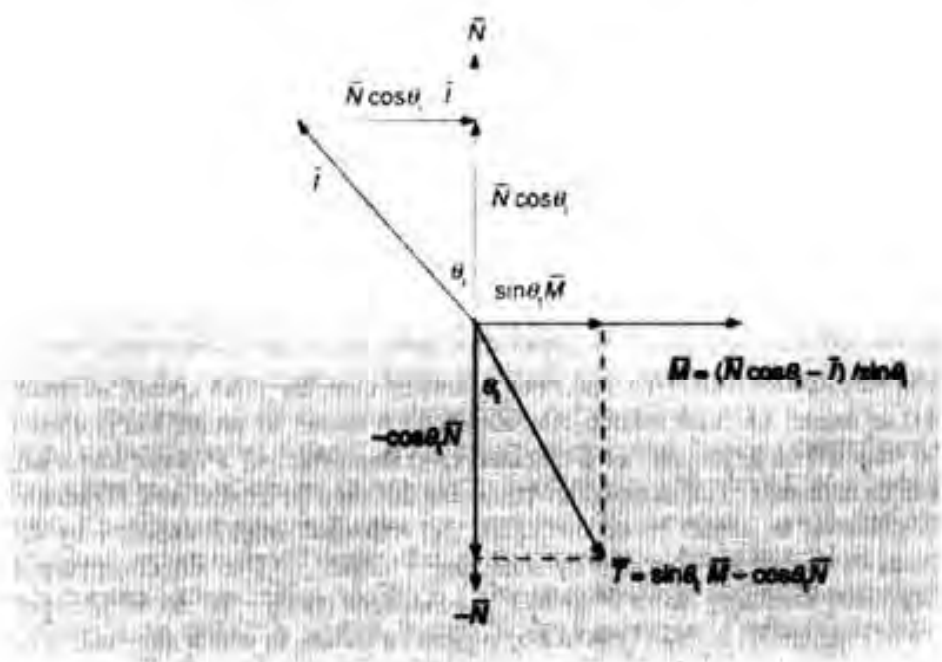


Fig. 16.36 Calculating the refraction vector.

Note that $\cos\theta_i$ is $\overline{N} \cdot \overline{I}$, and $\cos\theta_t$ can be computed as

$$\cos\theta_t = \sqrt{1 - \sin^2\theta_t} = \sqrt{1 - \eta_{r\lambda}^2 \sin^2\theta_i} = \sqrt{1 - \eta_{r\lambda}^2(1 - (\overline{N} \cdot \overline{I})^2)}. \quad (16.32)$$

Thus,

$$\overline{T} = \left(\eta_{r\lambda}(\overline{N} \cdot \overline{I}) - \sqrt{1 - \eta_{r\lambda}^2(1 - (\overline{N} \cdot \overline{I})^2)}\right)\overline{N} - \eta_{r\lambda}\overline{I}. \quad (16.33)$$

Total internal reflection. When light passes from one medium into another whose index of refraction is lower, the angle θ_t of the transmitted ray is greater than the angle θ_i. If θ_i becomes sufficiently large, then θ_t exceeds 90° and the ray is reflected from the interface between the media, rather than being transmitted. This phenomenon is known as *total internal reflection*, and the smallest θ_i at which it occurs is called the *critical angle*. You can observe total internal reflection easily by looking through the front of a filled fish tank and trying to see your hand through a side wall. When the viewing angle is greater than the critical angle, the only visible parts of your hand are those pressed firmly against the tank, with no intervening layer of air (which has a lower index of refraction than glass or water). The critical angle is the value of θ_i at which $\sin\theta_t$ is 1. If $\sin\theta_t$ is set to 1 in Eq. (16.28), we can see that the critical angle is $\sin^{-1}(\eta_{t\lambda} / \eta_{i\lambda})$. Total internal reflection occurs when the square root in Eq. (16.33) is imaginary.

Section 16.12 discusses the use of Snell's law in modeling refractive transparency with ray tracing; translucency is treated in Sections 16.12.4 and 16.13. An approximation of refraction can also be incorporated into renderers that proceed in back-to-front order [KAY79b].

16.6 INTEROBJECT REFLECTIONS

Interobject reflections occur when a surface reflects other surfaces in its environment. These effects range from more or less sharp specular reflections that change with the viewer's position (like specular highlights), to diffuse reflections that are insensitive to the viewer's position. Ray tracing (Section 16.12) and radiosity methods (Section 16.13) have produced some of the most visually impressive pictures exhibiting specular and diffuse interobject reflections; earlier techniques, however, can also produce attractive results.

Blinn and Newell [BLIN76] developed *reflection mapping* (also known as *environment mapping*) to model specular interobject reflection. A center of projection is chosen from which to map the environment to be reflected onto the surface of a sphere surrounding the objects to be rendered. The mapped environment can then be treated as a 2D texture map. At each point on an object to be displayed, the reflection map is indexed by the polar coordinates of the vector obtained by reflecting $\overline{V}$ about $\overline{N}$. The reflection map's x and y axes represent longitude (from 0° to 360°) and latitude (from −90° to 90°), respectively, as shown in Fig. 16.37(a). Hall [HALL86] suggests a variant in which the y axis is sin (latitude), so that equal areas on the sphere map to equal areas on the reflection map (Fig.

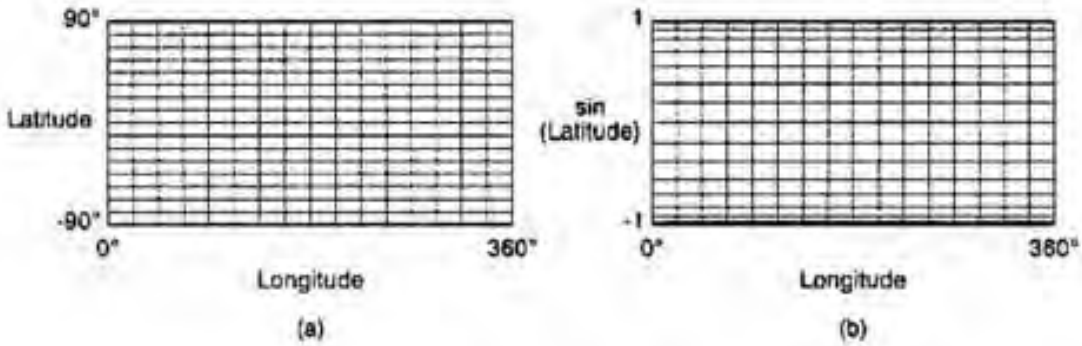
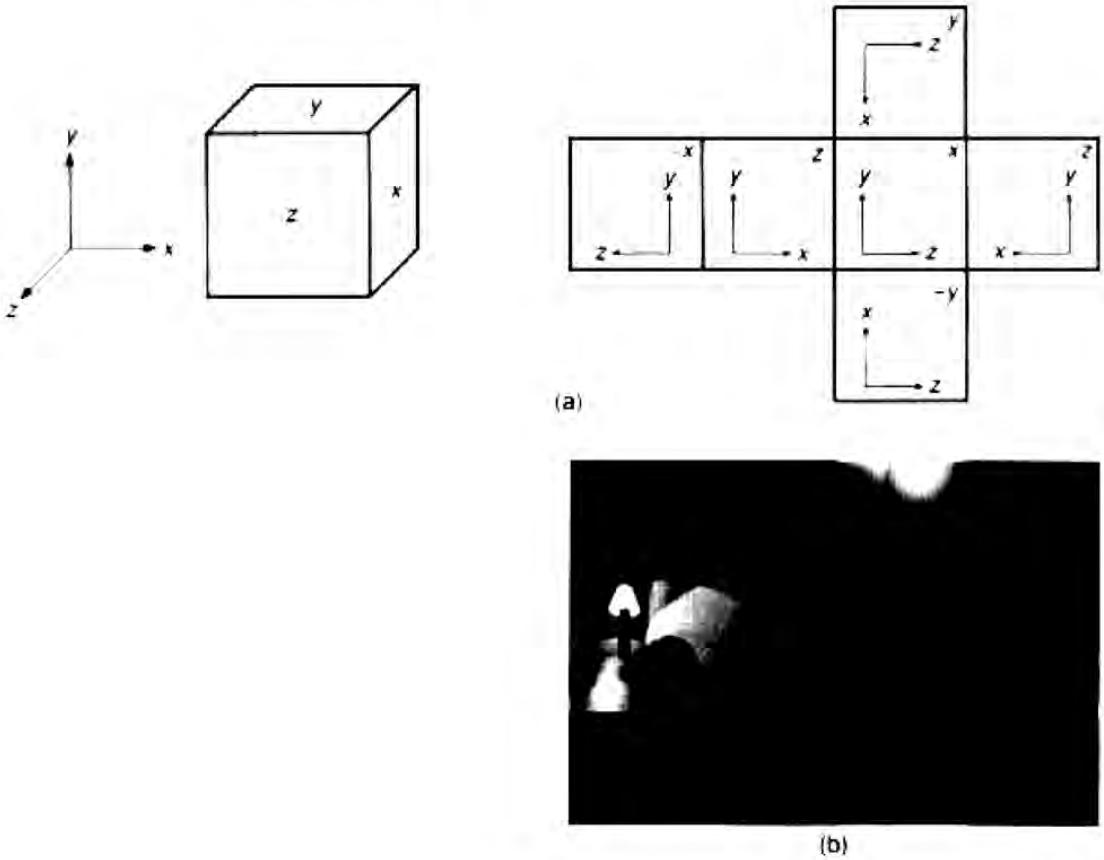


Fig. 16.37 Reflection map coordinate systems. (a) Latitude–longitude. (b) Sin (latitude)–longitude.

16.37b). Alternatively, six projections onto the sides of a surrounding cube may be used. The cube is aligned with the WC axes, so that the largest coordinate of the normalized reflection vector indicates the appropriate side to index. Figure 16.38(a) shows the correspondence between the unfolded sides and the cube; Fig. 16.38(b) is the reflection map used for the teapot in Color Plate II.37. As in texture mapping, antialiasing is accomplished by filtering some number of reflection-map values surrounding the indexed value to determine the reflected light at the given point on the object. In Fig. 16.38(b) an angle slightly wider than 90° was used to provide a margin about each side's borders that helps avoid the need to consider more than one side at a time when filtering.

Although reflection mapping can be used to produce a number of useful effects [GREE86], it provides only an approximation to the correct reflection information. By taking into account just the surface's reflection direction and not its position in the sphere, it models an infinitely large environment sphere. This problem can be partly remedied by using the surface's position to help determine the part of the reflection map to index, modeling a sphere of finite size. In either case, however, the farther a surface is from the center of projection used to create the map, the more distorted a view of the world it shows, since the reflection map takes into account visibility relationships at only a single point. A useful compromise is to create multiple reflection maps, each centered about a key object, and to index into the one closest to an object whose surface is being mapped. Simple but effective reflection effects can be obtained with even a 1D reflection map. For example, the y component of the reflection of $\overline{V}$ may be used to index into an array of intensities representing the range of colors from ground through horizon to sky.

Planar surfaces present difficulties for reflection mapping, because the reflection angle changes so slowly. If reflections from a planar surface are to be viewed from only a single viewpoint, however, another technique can be used. The viewpoint is reflected about the surface's plane and an inverted image of the scene is rendered from the reflected viewpoint, as shown in cross-section in Fig. 16.38(c). This image can then be merged with the original image wherever the surface is visible. Figure 16.38(d) was created with this technique and was used to render the reflections on the floor in Color Plate II.37



16.7 PHYSICALLY BASED ILLUMINATION MODELS

The illumination models discussed in the previous sections are largely the result of a common-sense, practical approach to graphics. Although the equations used approximate some of the ways light interacts with objects, they do not have a physical basis. In this section, we discuss physically based illumination models, relying in part on the work of Cook and Torrance [COOK82].

Thus far, we have used the word *intensity* without defining it, referring informally to the intensity of a light source, of a point on a surface, or of a pixel. It is time now to formalize our terms by introducing the radiometric terminology used in the study of thermal radiation, which is the basis for our understanding of how light interacts with objects [NICO77; SPARR78; SIEG81; IES87]. We begin with *flux*, which is the rate at which light energy is emitted and is measured in watts (W). To refer to the amount of flux emitted in or received from a given direction, we need the concept of a *solid angle*, which is the angle at the apex of a cone. Solid angle is measured in terms of the area on a sphere intercepted by a cone whose apex is at the sphere's center. A *steradian* (sr) is the solid angle of such a cone

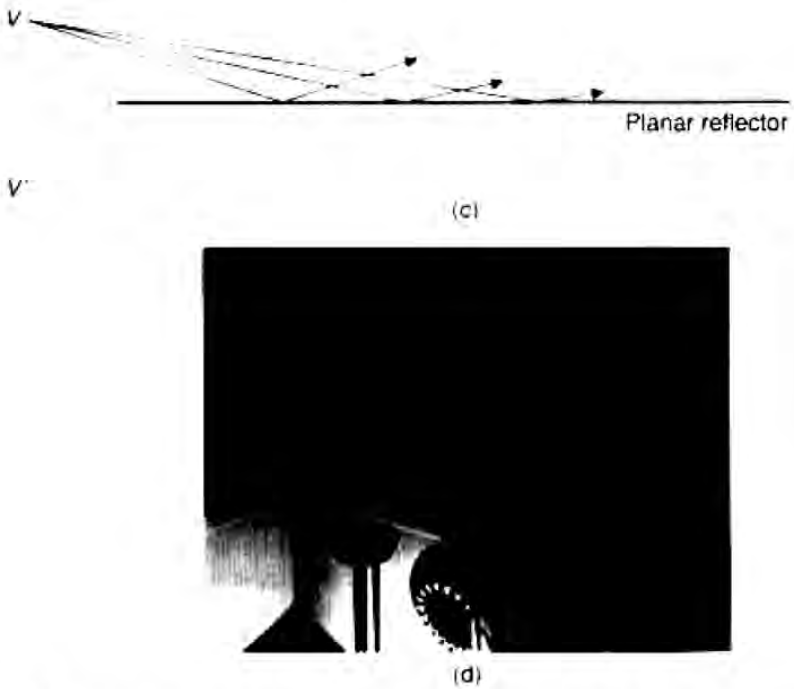


Fig. 16.38 Reflection maps. (a) Cube reflection map layout. (b) Reflection map for teapot in Color Plate II.37. (c) Geometry of planar reflection. (d) Reflected image merged with floor in Color Plate II.37. (Copyright © 1990, Pixar. Images in (b) and (d) rendered by Thomas Williams and H. B. Siegel using Pixar's PhotoRealistic RenderMan™ software.)

that intercepts an area equal to the square of the sphere's radius r. If a point is on a surface, we are concerned with the hemisphere above it. Since the area of a sphere is $4\pi r^2$, there are $4\pi r^2 / 2r^2 = 2\pi$ sr in a hemisphere. Imagine projecting an object's shape onto a hemisphere centered about a point on the surface that serves as the center of projection. The solid angle ω subtended by the object is the area on the hemisphere occupied by the projection, divided by the square of the hemisphere's radius (the division eliminates dependence on the size of the hemisphere). Thus, for convenience, we often speak of solid angle in terms of the area projected on a unit sphere or hemisphere, as shown in Fig. 16.39.

Radiant intensity is the flux radiated into a unit solid angle in a particular direction and is measured in W / sr. When we used the word *intensity* in reference to a point source, we were referring to its radiant intensity.

Radiance is the radiant intensity per unit foreshortened surface area, and is measured in W / (sr · m²). *Foreshortened surface area*, also known as *projected surface area*, refers to the projection of the surface onto the plane perpendicular to the direction of radiation. The foreshortened surface area is found by multiplying the surface area by $\cos \theta_r$, where θ_r is the

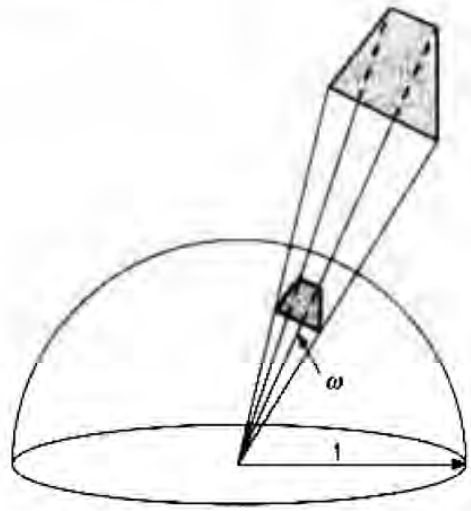


Fig. 16.39 The solid angle subtended by an object from a point on a surface is the area covered by the object's projection onto a unit hemisphere above the point.

angle of the radiated light relative to the surface normal. A small solid angle $d\omega$ may be approximated as the object's foreshortened surface area divided by the square of the distance from the object to the point at which the solid angle is being computed. When we used the word *intensity* in reference to a surface, we were referring to its radiance. Finally, *irradiance*, also known as *flux density*, is the incident flux per (unforeshortened) unit surface area and is measured in W / m^2.

In graphics, we are interested in the relationship between the light incident on a surface and the light reflected from and transmitted through that surface. Consider Fig. 16.40. The irradiance of the incident light is

$$E_i = I_i(\overline{N} \cdot \overline{L})\, d\omega_i, \tag{16.34}$$

where I_i is the incident light's radiance, and $\overline{N} \cdot \overline{L}$ is $\cos\theta_i$. Since irradiance is expressed per

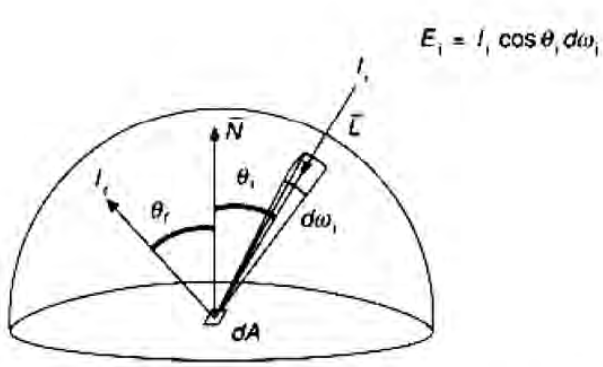


Fig. 16.40 Reflected radiance and incident irradiance.

unit area, whereas radiance is expressed per unit foreshortened area, multiplying by $\overline{N} \cdot \overline{L}$ converts it to the equivalent per unit unforeshortened area.

It is not enough to consider just I_i (the incident radiance) when determining I_r (the reflected radiance); E_i (the incident irradiance) must instead be taken into account. For example, an incident beam that has the same radiant intensity (W / sr) as another beam but a greater solid angle has proportionally greater E_i and causes the surface to appear proportionally brighter. The ratio of the reflected radiance (intensity) in one direction to the incident irradiance (flux density) responsible for it from another direction is known as the *bidirectional reflectivity*, ρ, which is a function of the directions of incidence and reflection,

$$\rho = \frac{I_r}{E_i}. \tag{16.35}$$

Thus, substituting for E_i from Eq. (16.34), we get

$$I_r = \rho I_i(\overline{N} \cdot \overline{L})d\omega_i. \tag{16.36}$$

The irradiance incident on a pixel in the image (the *image irradiance*) is proportional to the radiance emitted by the scene (the *scene radiance*) that is focused on the pixel [HORN79]. The factor of proportionality is a function of the imaging system being used.

As we have seen, it is conventional in computer graphics to consider bidirectional reflectivity as composed of diffuse and specular components. Therefore,

$$\rho = k_d\rho_d + k_s\rho_s, \tag{16.37}$$

where ρ_d and ρ_s are respectively the diffuse and specular bidirectional reflectivities, and k_d and k_s are respectively the diffuse and specular reflection coefficients introduced earlier in this chapter; $k_d + k_s = 1$. It is important to note that Eq. (16.37) is a useful approximation that is not applicable to all surfaces. For example, the lunar surface has a ρ that peaks in the direction of incidence [SIEG81]. During a full moon, when the sun, earth, and moon are nearly in line, this accounts for why the moon appears as a disk of roughly uniform intensity. If the moon had a Lambertian surface, it would, instead, reflect more light at its center than at its sides.

In addition to the effects of direct light-source illumination, we need to take into account illumination by light reflected from other surfaces. The lighting models discussed so far have assumed that this ambient light is equally incident from all directions, is independent of viewer position, and is not blocked by any nearby objects. Later in this chapter we shall discuss how to lift these restrictions. For now we retain them, modeling the ambient term as $\rho_a I_a$, where ρ_a is the nondirectional ambient reflectivity, and I_a is the incident ambient intensity.

The resulting illumination equation for n light sources is

$$I_r = \rho_a I_a + \sum_{1 \le j \le n} I_{i_j}(\overline{N} \cdot \overline{L}_j)\, d\omega_{i_j}(k_d\rho_d + k_s\rho_s). \tag{16.38}$$

To reinterpret the illumination model of Eq. (16.15) in terms of Eq. (16.38), we expressed diffuse reflectivity as the object's diffuse color, and specular reflectivity using the product of the object's specular color and a $\cos^n \alpha$ term. We have already noted some of the

inadequacies of that specular reflectivity formulation. Now we shall examine how to replace it.

16.7.1 Improving the Surface Model

The Torrance–Sparrow model [TORR66; TORR67], developed by applied physicists, is a physically based model of a reflecting surface. Blinn was the first to adapt the Torrance–Sparrow model to computer graphics, giving the mathematical details and comparing it to the Phong model in [BLIN77a]; Cook and Torrance [COOK82] were the first to approximate the spectral composition of reflected light in an implementation of the model.

In the Torrance–Sparrow model, the surface is assumed to be an isotropic collection of planar microscopic facets, each a perfectly smooth reflector. The geometry and distribution of these *microfacets* and the direction of the light (assumed to emanate from an infinitely distant source, so that all rays are parallel) determine the intensity and direction of specular reflection as a function of I_p (the point light source intensity), $\bar{N}$, $\bar{L}$, and $\bar{V}$. Experimental measurements show a very good correspondence between the actual reflection and the reflection predicted by this model [TORR67].

For the specular component of the bidirectional reflectivity, Cook and Torrance use

$$\rho_s = \frac{F_\lambda}{\pi} \frac{DG}{(\bar{N} \cdot \bar{V})(\bar{N} \cdot \bar{L})}, \tag{16.39}$$

where D is a distribution function of the microfacet orientations, G is the *geometrical attenuation factor*, which represents the masking and shadowing effects of the microfacets on each other, and F_λ is the Fresnel term computed by Fresnel's equation (described later), which, for specular reflection, relates incident light to reflected light for the smooth surface of each microfacet. The π in the denominator is intended to account for surface roughness (but see [JOY88, pp. 227–230] for an overview of how the equation is derived). The $\bar{N} \cdot \bar{V}$ term makes the equation proportional to the surface area (and hence to the number of microfacets) that the viewer sees in a unit piece of foreshortened surface area, whereas the $\bar{N} \cdot \bar{L}$ term makes the equation proportional to the surface area that the light sees in a unit piece of foreshortened surface area.

16.7.2 The Microfacet Distribution Function

Since the microfacets are considered to be perfect specular reflectors, the model considers only those microfacets whose normals lie along the halfway vector $\bar{H}$, introduced in Section 16.1.4. Only a fraction D of the microfacets have this orientation. Torrance and Sparrow assumed a Gaussian distribution function for D in their original work. Blinn used the Trowbridge and Reitz distribution [TROW75], and Cook and Torrance used the Beckmann distribution function [BECK63]. Cook and Torrance point out the Beckmann distribution has a good theoretical basis and has no arbitrary constants, unlike the distributions used by Torrance and Sparrow, and by Blinn. The Beckmann distribution function for rough surfaces is

$$D = \frac{1}{4m^2 \cos^4\beta} e^{-((\tan\beta)/m)^2}, \tag{16.40}$$

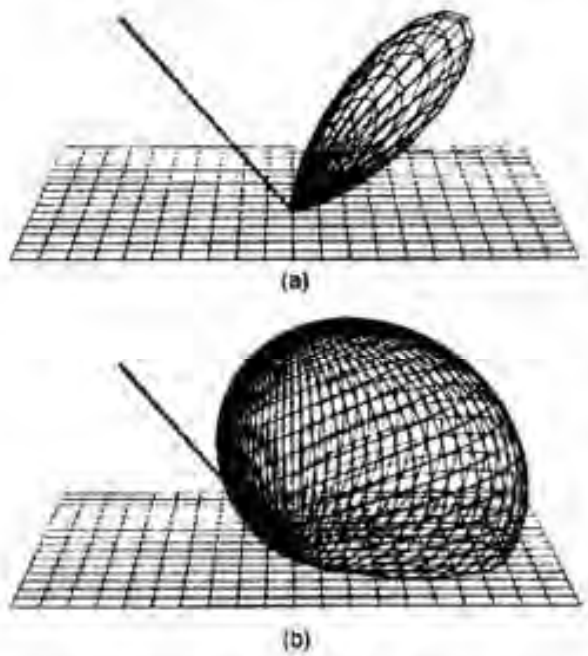


Fig. 16.41 Beckmann microfacet distribution function for (a) $m = 0.2$ and (b) $m = 0.6$. (From [COOK82] with permission. By Robert Cook, Program of Computer Graphics, Cornell University.)

where β is the angle between $\overline{N}$ and $\overline{H}$, and m is the root-mean-square slope of the microfacets.[1] When m is small, the microfacet slopes vary only slightly from the surface normal and, as we would expect, the reflection is highly directional (Fig. 16.41a). When m is large, the microfacet slopes are steep and the resulting rough surface spreads out the light it reflects (Fig. 16.41b). To model surfaces that have multiple scales of roughness, Cook and Torrance use a weighted sum of distribution functions,

$$D = \sum_{1 \le j \le n} w_j D(m_j), \tag{16.41}$$

where the sum of the weights w_j is 1.

16.7.3 The Geometrical Attenuation Factor

The model takes into account that some microfacets may shadow others. Torrance and Sparrow and Blinn discuss the calculation of G, considering three different situations. Figure 16.42(a) shows a microfacet whose incident light is totally reflected. Figure 16.42(b) shows a microfacet that is fully exposed to the rays, but has some reflected rays that are shielded by other microfacets. These shielded rays ultimately contribute to the diffuse reflection. Figure 16.42(c) shows a microfacet that is partially shielded from the light. The geometric attenuation factor G ranges from 0 (total shadowing) to 1 (no shadowing).

[1]Hall [HALL89] mentions that [COOK82] is missing the 4 in the denominator.

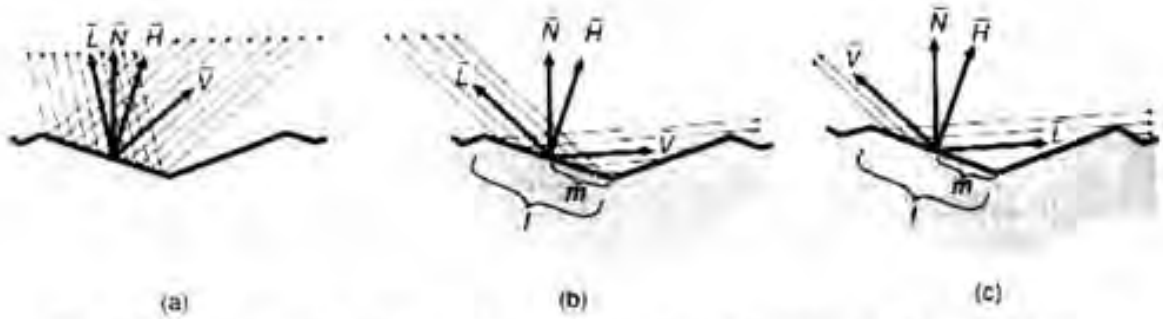


Fig. 16.42 Light rays reflecting from surface microfacets in the Torrance-Sparrow model. (a) No interference. (b) Partial interception of reflected light. (c) Partial interception of incident light. (After [BLIN77a].)

To simplify the analysis, the microfacets are assumed to form V-shaped grooves that are symmetric about the average surface normal $\overline{N}$. In Fig. 16.42(a), all the incident light is reflected to the viewer; thus, G_a is 1. In both other cases, the ratio of intercepted light is given by m / l, where l is the total area of the facet and m is the area whose reflected light is blocked (Fig. 16.42b) or that is itself blocked (Fig 16.42c) from the light. Therefore, $G_b = G_c = 1 - m / l$. Blinn gives a trigonometric derivation of the proportion of reflected light in Fig. 16.42(b) as

$$G_b = \frac{2(\overline{N} \cdot \overline{H})(\overline{N} \cdot \overline{V})}{(\overline{V} \cdot \overline{H})}. \tag{16.42}$$

The ratio for the case in Fig. 16.42(c) follows by noticing that it is the same as the case in part (b), except that $\overline{L}$ and $\overline{V}$ trade places:

$$G_c = \frac{2(\overline{N} \cdot \overline{H})(\overline{N} \cdot \overline{L})}{(\overline{V} \cdot \overline{H})}. \tag{16.43}$$

The denominator does not need to change because $(\overline{V} \cdot \overline{H}) = (\overline{L} \cdot \overline{H})$ by definition of $\overline{H}$ as the halfway vector between $\overline{V}$ and $\overline{L}$.

G is the minimum of the three values:

$$G = \min\left\{1, \frac{2(\overline{N} \cdot \overline{H})(\overline{N} \cdot \overline{V})}{(\overline{V} \cdot \overline{H})}, \frac{2(\overline{N} \cdot \overline{H})(\overline{N} \cdot \overline{L})}{(\overline{V} \cdot \overline{H})}\right\}. \tag{16.44}$$

16.7.4 The Fresnel Term

The Fresnel equation for unpolarized light specifies the ratio of reflected light from a dielectric (nonconducting) surface as

$$F_\lambda = \frac{1}{2}\left(\frac{\tan^2(\theta_i - \theta_t)}{\tan^2(\theta_i + \theta_t)} + \frac{\sin^2(\theta_i - \theta_t)}{\sin^2(\theta_i + \theta_t)}\right) = \frac{1}{2}\frac{\sin^2(\theta_i - \theta_t)}{\sin^2(\theta_i + \theta_t)}\left(1 + \frac{\cos^2(\theta_i + \theta_t)}{\cos^2(\theta_i - \theta_t)}\right), \tag{16.45}$$

where θ_i is the angle of incidence relative to $\overline{H}$ (i.e., $\cos^{-1}(\overline{L} \cdot \overline{H})$), and, as before, θ_t is the angle of refraction; $\sin \theta_t = (\eta_{i\lambda} / \eta_{t\lambda}) \sin \theta_i$, where $\eta_{i\lambda}$ and $\eta_{t\lambda}$ are the indices of refraction of the two media. The Fresnel equation can also be expressed as

$$F_\lambda = \frac{1}{2} \frac{(g - c)^2}{(g + c)^2} \left(1 + \frac{[c(g + c) - 1]^2}{[c(g - c) + 1]^2} \right), \tag{16.46}$$

where $c = \cos \theta_i = \overline{L} \cdot \overline{H}$, $g^2 = \eta_\lambda^2 + c^2 - 1$, and $\eta_\lambda = \eta_{t\lambda} / \eta_{i\lambda}$.

In a conducting medium, which attenuates light, it is necessary to refer to $\overline{\eta}_\lambda$, the material's *complex index of refraction*, defined as

$$\overline{\eta}_\lambda = \eta_\lambda - i\kappa_\lambda, \tag{16.47}$$

where κ_λ is the material's *coefficient of extinction*, which measures the amount that the material attenuates intensity per unit path length. To simplify the reflection computations for conductors, κ_λ can be assumed to be zero and a single equivalent real value for η_λ can be determined.

Blinn created Figs. 16.43 and 16.44, comparing the effects of the Phong illumination model and the Torrance–Sparrow model. He made the simplifying assumptions that the specular term depends on only the color of the incident light, and that the viewer and light source are both at infinity. Figures 16.43 and 16.44 show the reflected illumination from a surface for angles of incidence of 30° and 70°, respectively. In each figure, the vertical arrow represents the surface normal, the incoming arrow represents the direction of light rays, and the outgoing arrow represents the direction of reflection for a perfect reflector. The rounded part of each figure is the diffuse reflection, whereas the bump is the specular reflection. For the 30° case in Fig. 16.43, the models produce nearly similar results, but for the 70° case in Fig. 16.44, the Torrance–Sparrow model has much higher specular reflectance and the peak occurs at an angle greater than the angle of incidence. This so-called *off-specular peak* is observed in actual environments. Figure 16.45, also by Blinn, shows a marked difference in the visual effect of the two models as the light source moves away from the viewpoint to the side and then to the rear of a metallic sphere.

(a) Phong model (b) Torrance-Sparrow model

Fig. 16.43 Comparison of Phong and Torrance–Sparrow illumination models for light at a 30° angle of incidence. (By J. Blinn [BLIN77a], courtesy of the University of Utah.)

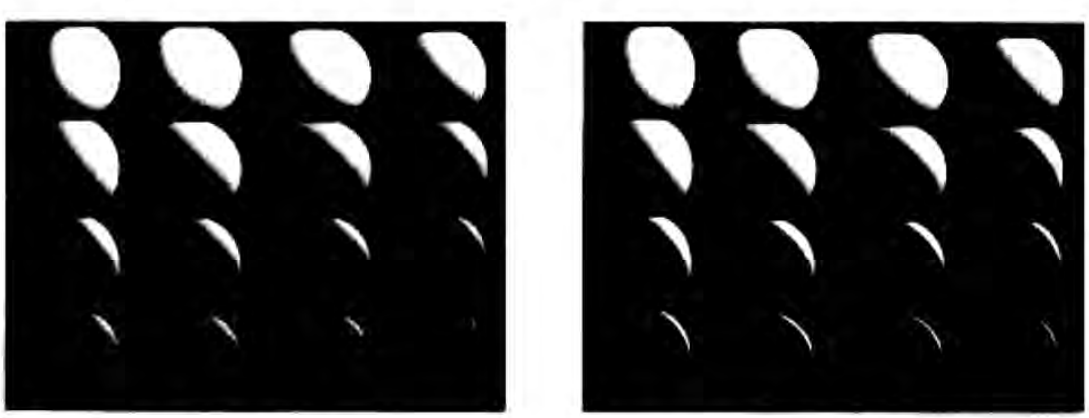
(a) Phong model (b) Torrance-Sparrow model

Fig. 16.44 Comparison of Phong and Torrance–Sparrow illumination models for light at a 70° angle of incidence. (By J. Blinn [BLIN77a], courtesy of the University of Utah.)

The specular-reflection color shift. Like Blinn, Cook and Torrance use the Torrance-Sparrow surface model to determine the specular term. Unlike Blinn, however, they make the color of the specular reflection a function of the interaction between the material and the incident light, depending both on the light's wavelength and on its angle of incidence. This is correct because the Fresnel equation, Eq. (16.45), is responsible for a shift in the specular reflection color based on the angle the incident light makes with the microfacet normal $\overline{H}$, as shown in Fig. 16.46.

When the incident light is in the same direction as $\overline{H}$, then $\theta_i = 0$, so $c = 1$ and $g = \eta_\lambda$. Substituting these values in Eq. (16.46) yields the Fresnel term for $\theta_i = 0$,

$$F_{\omega} = \left(\frac{\eta_\lambda - 1}{\eta_\lambda + 1}\right)^2. \tag{16.48}$$



(a) Phong model (b) Torrance-Sparrow model

Fig. 16.45 Comparison of Phong and Torrance–Sparrow illumination models for a metallic sphere illuminated by a light source from different directions. Differences are most apparent for back-lit cases (bottom rows). (By J. Blinn [BLIN77a], courtesy of the University of Utah.)

Fig. 16.46 Fresnel term for a copper mirror as a function of wavelength and angle of incidence. (By Robert Cook, Program of Computer Graphics, Cornell University.)

When the incident light grazes the surface of the microfacet, then $\theta_i = \pi / 2$, so $c = 0$. Substituting in Eq. (16.46) yields the Fresnel term for $\theta_i = \pi / 2$,

$$F_{\lambda\pi/2} = 1. \tag{16.49}$$

Thus, if light is normal to the surface, then $F_{\lambda 0}$, and hence the specular reflectance ρ_s, are functions of the surface's index of refraction, which in turn varies with the wavelength. When the light grazes the surface (and when the viewer is looking 180° opposite from the light, since only microfacets with normals of $\overline{H}$ are considered), $F_{\lambda\pi/2}$, and hence the specular reflection, are both 1. Specular reflectance depends on η_λ for all θ_i except $\pi / 2$. For metals, essentially all reflection occurs at the surface and is specular. Only at extreme glancing angles is the specular reflection not influenced by the object's material. Notice how this differs from the Phong specular-reflection term, which was always unaffected by the object color. Cook and Torrance point out that the monochromatic Phong specular term is a good model of plastic, which is colored by pigment particles embedded in a transparent substrate. The presence of these particles causes the diffuse reflection, resulting from light bouncing around in the substrate, to be a function of the pigment and light colors. Specular reflection occurs from the transparent surface, however, and therefore is unaffected by the pigment color. This is why objects rendered with a Phong illumination model look plastic.

If the indices of refraction at different wavelengths are known, they may be used directly in the Fresnel equation. More typically, however, they are not known. Reflectance, however, has been measured for many materials with $\theta_i = 0$ at a variety of wavelengths, as recorded in sources such as [TOUL70; TOUL72a; TOUL72b]. In this case, each η_λ may be determined from Eq. (16.48), as

$$\eta_\lambda = \frac{1 + \sqrt{F_{\lambda 0}}}{1 - \sqrt{F_{\lambda 0}}}. \tag{16.50}$$

A derived value of η_λ may then be used in the Fresnel equation to determine F_λ for an arbitrary θ_i. Rather than perform this calculation for each value of λ, Cook and Torrance simplify the computationally expensive color shift calculation by calculating $F_{\text{avg}\eta_\lambda}$ for $\eta_{\text{avg}\lambda}$,

the average index of refraction for average normal reflectance. They use the value computed for $F_{\text{avg}\theta_i}$ to interpolate between the color of the material at $\theta_i = 90°$ and the color of the material at $\theta_i = 0°$ for each component of the color model. Because $F_{\lambda\pi/2}$ is always 1, the color of the material at $\theta_i = 90°$ is the color of the light source. Thus, when the light grazes the surface at a 90° angle of incidence, the color of the reflected light is that of the incident light. Using the RGB system, we call the red component of the material at $\theta_i = 0°$, Red_0, and the red component of the light, $\text{Red}_{\pi/2}$. Red_0 is calculated by integrating the product of F_0, the spectrum of the incident light, and the color-matching curves of Fig. 13.22, and applying the inverse of matrix M of Eq. (13.24) to the result. $\text{Red}_{\pi/2}$ is obtained by applying the inverse of M to Eq. (13.18). The approximation computes the color of the material at θ_i as

$$\text{Red}_{\theta i} = \text{Red}_0 + (\text{Red}_{\pi/2} - \text{Red}_0)\,\frac{\max(0,\ F_{\text{avg}\theta_i} - F_{\text{avg}0})}{F_{\text{avg}\pi/2} - F_{\text{avg}0}}. \tag{16.51}$$

$\text{Red}_{\theta i}$ is then used in place of F_λ in Eq. (16.39). Because the approximation takes the light's spectrum into account, Eq. (16.38) must be modified to multiply the specular term by a wavelength-independent intensity scale factor, instead of by the light's spectrum. Hall [HALL89] suggests an alternative approximation that interpolates a value for $F_{\lambda\theta_i}$, given $F_{\lambda 0}$. Since $F_{\lambda\pi/2}$ is always 1 (as is $F_{\text{avg}\pi/2}$ in Eq. 16.51),

$$F_{\lambda\theta_i} = F_{\lambda 0} + (1 - F_{\lambda 0})\,\frac{\max(0,\ F_{\text{avg}\theta_i} - F_{\text{avg}0})}{1 - F_{\text{avg}0}}. \tag{16.52}$$

Color Plate III.8 shows two copper vases rendered with the Cook–Torrance model, both of which use the bidirectional reflectance of copper for the diffuse term. The first models the specular term using the reflectance of a vinyl mirror and represents results similar to those obtained with the original Phong illumination model of Eq. (16.14). The second models the specular term with the reflectance of a copper mirror. Note how accounting for the dependence of the specular highlight color on both angle of incidence and surface material produces a more convincing image of a metallic surface.

In general, the ambient, diffuse, and specular components are the color of the material for both dielectrics and conductors. Composite objects, such as plastics, typically have diffuse and specular components that are different colors. Metals typically show little diffuse reflection and have a specular component color that ranges between that of the metal and that of the light source as θ_i approaches 90°. This observation suggests a rough approximation to the Cook–Torrance model that uses Eq. (16.15) with $O_{s\lambda}$ chosen by interpolating from a look-up table based on θ_i.

Further work. Kajiya [KAJI85] has generalized the Cook–Torrance illumination model to derive *anisotropic illumination models* whose reflective properties are not symmetric about the surface normal. These more accurately model the way that light is reflected by hair or burnished metal—surfaces whose microfeatures are preferentially oriented. To do this, Kajiya extends bump mapping to perturb not just the surface normal, but also the

tangent and a binormal formed by the cross-product of the tangent and normal. Together these form a coordinate system that determines the orientation of the surface relative to the anisotropic illumination model. The surface shown in Fig. 16.47 is mapped to an anisotropic texture that represents a cross-weave of threads. Cabral, Max and Springmeyer [CABR87] have developed a method for determining G and ρ for a surface finish specified by an arbitrary bump map by computing the shadowing and masking effects of the bumps.

The Fresnel equation used by Blinn and by Cook and Torrance is correct only for unpolarized light. The polarization state of light changes, however, when light is reflected from a surface, and a surface's ρ is a function of the polarization state of the light incident on it. Wolff and Kurlander [WOLF90] have extended the Cook–Torrance model to take this into account and have generated images that evidence two effects that are most visible after two or more interobject reflections. First, dielectrics have an angle of incidence, known as the *Brewster angle*, at which incident light is completely polarized when reflected, or is not reflected at all if it is inappropriately polarized. If interobject specular reflection of initially unpolarized light occurs between two dielectric surfaces such that the angle of incidence of each reflection is equal to the Brewster angle, and the plane defined by $\overline{N}$ and $\overline{L}$ on one surface is perpendicular to that of the other, then no light at all will be specularly reflected from the second object; we can produce noticeable, but less dramatic, attenuation if the angles and orientations are varied. Second, colored conductors (metals such as copper or gold) tend to polarize light at different wavelengths differently. Therefore, when a colored conductor is reflected from a dielectric surface, the reflection will have a color slightly different from that when polarization is not taken into account.

Fig. 16.47 Anisotropic texture. (By J. Kajiya [KAJI85], California Institute of Technology.)

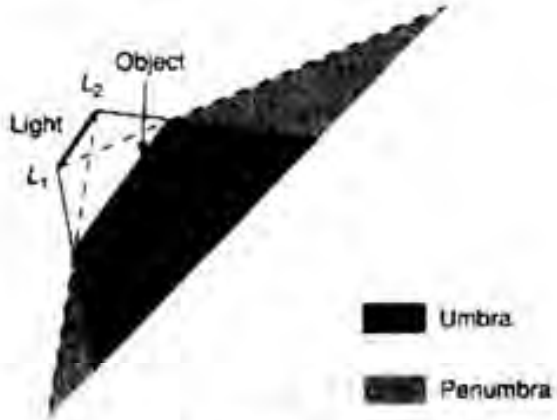


Fig. 16.48 Umbra and penumbra.

16.8 EXTENDED LIGHT SOURCES

The light sources discussed thus far have been point lights. In contrast, *extended* or *distributed* light sources actually have area and consequently cast "soft" shadows containing areas only partially blocked from the source, as shown in Fig. 16.48. The part of a light source's shadow that is totally blocked from the light source is the shadow's *umbra*;

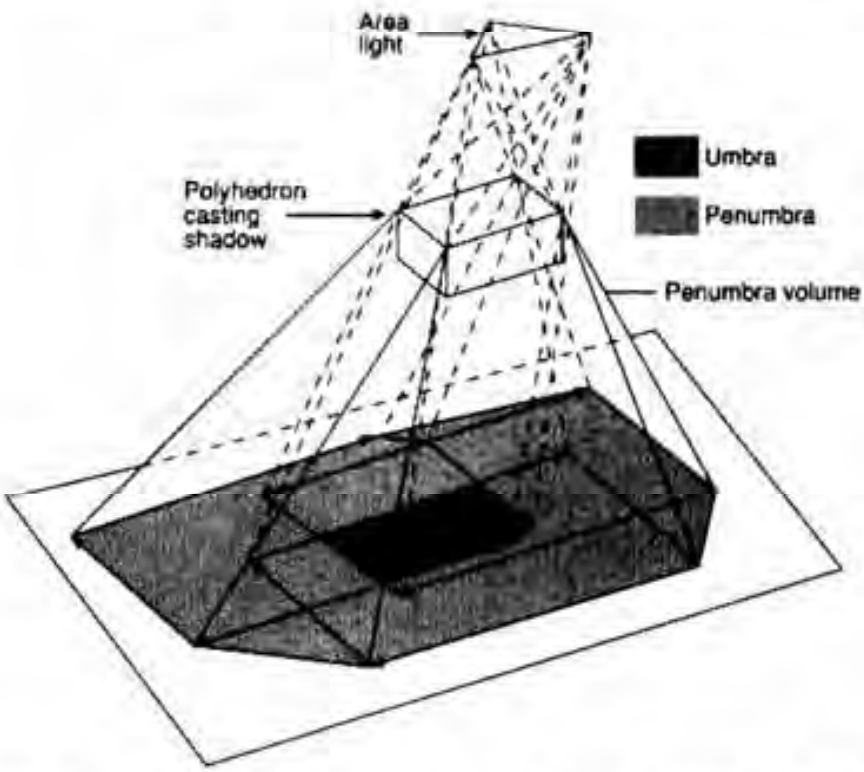


Fig. 16.49 Determining the penumbra and umbra volumes. (After [NISH85a].)

that part of the shadow that is only partially shielded from the source is the shadow's *penumbra*. All of a point light source's shadow is umbra. An obvious approach to modeling an extended light source is to approximate it with closely clustered point light sources [VERB84; BROT84]. The process is computationally expensive, however, and the results are less than satisfactory unless there are very many point sources and no specular reflection.

Nishita and Nakamae [NISH83; NISH85a; NISH85b] have developed an extension of shadow volumes for modeling linear, convex polygonal, and convex polyhedral light sources with Lambertian intensity distributions. Their method employs an object-precision algorithm for determining shadow volumes for convex polyhedral objects. As shown in Fig. 16.49, the shadow volumes defined by each vertex of the source and the polyhedron are determined. The penumbra volume is then the smallest convex polyhedron containing these shadow volumes (their convex hull), and the umbra volume is the intersection of the shadow volumes. The volumes are then intersected with the faces of the objects to determine the areas in each penumbra or umbra volume. Any point lying within an umbra volume defined by a light source and any other polyhedron is not affected by the light source. Determining the color of a point in a penumbra involves computing those parts of the light source visible from the point on the object. A BSP-tree–based approach is discussed in [CHIN90].

Excellent simulations of extended light sources have been achieved by using variations on ray-tracing (Section 16.12) and radiosity methods (Section 16.13).

16.9 SPECTRAL SAMPLING

As light is reflected from a surface, the spectral energy distribution P_λ of the incident light is modified by the spectral reflectance function of the surface, ρ_λ. This curve specifies the percentage of light of each wavelength λ that is reflected. Therefore, the spectral energy distribution of the reflected light is $P_\lambda \rho_\lambda$. Similarly, light passing through a material is modified by the spectral transmittance function of the material, τ_λ. Once a final spectral energy distribution for light falling on the eye at a particular point (i.e., for a particular pixel) is determined, then Eq. (13.18) can be used to find the corresponding CIE XYZ specification for the light, and this result can be converted to RGB using the inverse of M of Eq. (13.24).

It is tempting to assume that ρ_λ and τ_λ can be replaced by equivalent tristimulus RGB or XYZ color specifications. If this could be done, then the product $P_\lambda \rho_\lambda$ could be replaced by a sum of the tristimulus values for each of P_λ and ρ_λ, of the form

$$R\rho_R + G\rho_G + B\rho_B. \tag{16.53}$$

This is clearly incorrect. Consider the P_λ and ρ_λ shown in Fig. 16.50. P_λ is uniform except for a narrow interval around 600 nanometers, whereas ρ_λ reflects light only at the same narrow interval. The product $P_\lambda \rho_\lambda$ is thus zero everywhere, and the surface appears black. On the other hand, the XYZ components of P_λ will be nonzero (this can be seen by applying the inverse of M to the results of Eq. (13.18). Similarly, substituting ρ_λ for P_λ in Eq. (13.18) and applying the inverse of M gives nonzero values for ρ_R, ρ_G, ρ_B, so the product (Eq. 16.53) is also nonzero! This is the wrong answer, and demonstrates that ρ_λ and τ_λ cannot in general be replaced by an equivalent tristimulus color specification. As Hall

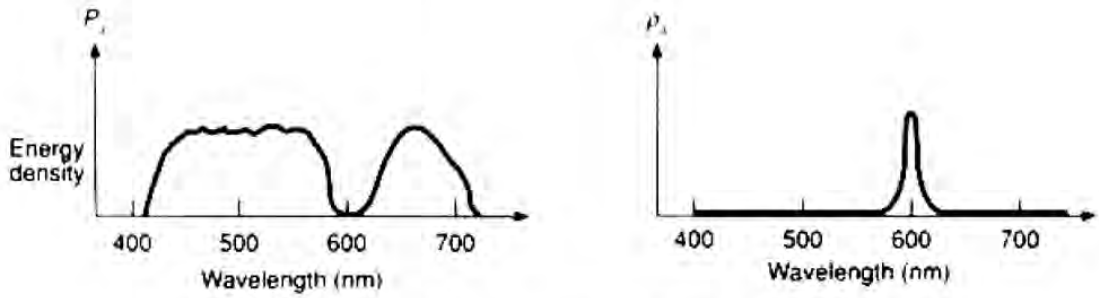


Fig. 16.50 P_λ and ρ_λ whose product is 0 everywhere.

[HALL83] points out, this approach fails because it is actually point sampling in color space and consequently is prone to all the problems that arise from undersampling.

A more accurate representation of a light source's spectral distribution or a surface's reflectivity or transmissivity can be obtained by representing either one as a curve that interpolates a sufficiently large set of samples spaced across the visible spectrum. Color Plate III.9 shows images, generated with different color models, of two overlapping filters lit by a D6500 light source. The filters do not pass a common band of wavelengths, so no transmitted light should be visible where they overlap. Color Plate III.9(a), the control picture, was generated using spectra maintained at 1-nm intervals from 360 to 830 nanometers. Color Plates III.9(b) and III.9(c) were generated with three samples for the primaries of the CIE and RGB color spaces, respectively. Color Plate III.9(d) approximates the spectra of Color Plate III.9(a) with nine spectral values. Note how much more closely Color Plate III.9(d) matches the control picture than do the others.

16.10 IMPROVING THE CAMERA MODEL

Thus far, we have modeled the image produced by a pinhole camera: each object, regardless of its position in the environment, is projected sharply and without distortion in the image. Real cameras (and eyes) have lenses that introduce a variety of distortion and focusing effects.

Depth of field. Objects appear to be more or less in focus depending on their distance from the lens, an effect known as *depth of field*. A lens has a focal length F corresponding to the distance from the lens at which a perfectly focused image of an object converges. If a point is out of focus, its image converges on a plane that is closer or farther than F. An out-of-focus point projects on a plane at F as a circle known as the *circle of confusion*.

Potmesil and Chakravarty [POTM82] have developed a postprocessing technique for simulating some of the effects of depth of field and other properties of real lenses, demonstrated in Color Plate II.38. Their system first produces images using a conventional pinhole-lens renderer that generates not only the intensity at each point, but also that point's z value. Each sampled point is then turned into a circle of confusion with a size and intensity distribution determined by its z value and the lens and aperture being used. The intensity of each pixel in the output image is calculated as a weighted average of the intensities in the circles of confusion that overlap the pixel. Since the image is initially computed from a single point at the center of projection, the results of this technique are

still only an approximation. A real lens's focusing effect causes light rays that would not pass through the pinhole to strike the lens and to converge to form the image. These rays see a slightly different view of the scene, including, for example, parts of surfaces that are not visible to the rays passing through the pinhole. This information is lost in the images created with the Potmesil and Chakravarty model.

Motion blur. *Motion blur* is the streaked or blurred appearance that moving objects have because a camera's shutter is open for a finite amount of time. To achieve this effect, we need to solve the visible-surface problem over time, as well as over space, to determine which objects are visible at a given pixel and when they are visible. Korein and Badler [KORE83] describe two contrasting approaches: an analytic algorithm that uses continuous functions to model the changes that objects undergo over time, and a simple image-precision approach that relies on temporal supersampling. In the temporal-supersampling method, a separate image is rendered for each point in time to be sampled. The motion-blurred image is created by taking a weighted sum of the images, in essence convolving them with a temporal filter. For example, if each of n images is weighted by $1/n$, this corresponds to temporal box filtering. The more closely the samples are spaced, the better the results. Temporal supersampling, like spatial supersampling, suffers from aliasing: Unless samples are spaced sufficiently close together in time, the final image will appear to be a set of discrete multiple exposures. Potmesil and Chakravarty [POTM83] have extended their depth-of-field work to model the ways in which actual camera shutters move. As we shall see in Section 16.12, the stochastic sampling techniques used in distributed ray tracing offer a uniform framework for integrating lens effects, motion blur, and spatial antialiasing demonstrated in Color Plates II.39 and III.16.

16.11 GLOBAL ILLUMINATION ALGORITHMS

An illumination model computes the color at a point in terms of light directly emitted by light sources and of light that reaches the point after reflection from and transmission through its own and other surfaces. This indirectly reflected and transmitted light is often called *global illumination*. In contrast, *local illumination* is light that comes directly from the light sources to the point being shaded. Thus far, we have modeled global illumination by an ambient illumination term that was held constant for all points on all objects. It did not depend on the positions of the object or the viewer, or on the presence or absence of nearby objects that could block the ambient light. In addition, we have seen some limited global illumination effects made possible by shadows, transparency, and reflection maps.

Much of the light in real-world environments does not come from direct light sources. Two different classes of algorithms have been used to generate pictures that emphasize the contributions of global illumination. Section 16.12 discusses extensions to the visible-surface ray-tracing algorithm that interleave visible-surface determination and shading to depict shadows, reflection, and refraction. Thus, global specular reflection and transmission supplement the local specular, diffuse, and ambient illumination computed for a surface. In contrast, the radiosity methods discussed in Section 16.13 completely separate shading and visible-surface determination. They model all an environment's interactions with light sources first in a view-independent stage, and then compute one or more images

for the desired viewpoints using conventional visible-surface and interpolative shading algorithms.

The distinction between view-dependent algorithms, such as ray tracing, and view-independent ones, such as radiosity, is an important one. *View-dependent* algorithms discretize the view plane to determine points at which to evaluate the illumination equation, given the viewer's direction. In contrast, *view-independent* algorithms discretize the environment, and process it in order to provide enough information to evaluate the illumination equation at any point and from any viewing direction. View-dependent algorithms are well-suited for handling specular phenomena that are highly dependent on the viewer's position, but may perform extra work when modeling diffuse phenomena that change little over large areas of an image, or between images made from different viewpoints. On the other hand, view-independent algorithms model diffuse phenomena efficiently, but require overwhelming amounts of storage to capture enough information about specular phenomena.

Ultimately, all these approaches attempt to solve what Kajiya [KAJI86] has referred to as the *rendering equation*, which expresses the light being transferred from one point to another in terms of the intensity of the light emitted from the first point to the second and the intensity of light emitted from all other points that reaches the first and is reflected from the first to the second. The light transferred from each of these other points to the first is, in turn, expressed recursively by the rendering equation. Kajiya presents the rendering equation as

$$I(x, x') = g(x, x')\left[\varepsilon(x, x') + \int_S \rho(x, x', x'')\, I(x', x'')\, dx''\right], \qquad (16.54)$$

where x, x', and x'' are points in the environment. $I(x, x')$ is related to the intensity passing from x' to x. $g(x, x')$ is a geometry term that is 0 when x and x' are occluded from each other, and $1/r^2$ when they are visible to each other, where r is the distance between them. $\varepsilon(x, x')$ is related to the intensity of light that is emitted from x' to x. The initial evaluation of $g(x, x')\varepsilon(x, x')$ for x at the viewpoint accomplishes visible-surface determination in the sphere about x. The integral is over all points on all surfaces S. $\rho(x, x', x'')$ is related to the intensity of the light reflected (including both specular and diffuse reflection) from x'' to x from the surface at x'. Thus, the rendering equation states that the light from x' that reaches x consists of light emitted by x' itself and light scattered by x' to x from all other surfaces, which themselves emit light and recursively scatter light from other surfaces.

As we shall see, how successful an approach is at solving the rendering equation depends in large part on how it handles the remaining terms and the recursion, on what combinations of diffuse and specular reflectivity it supports, and on how well the visibility relationships between surfaces are modeled.

16.12 RECURSIVE RAY TRACING

In this section, we extend the basic ray-tracing algorithm of Section 15.10 to handle shadows, reflection, and refraction. This simple algorithm determined the color of a pixel at the closest intersection of an eye ray with an object, by using any of the illumination models described previously. To calculate shadows, we fire an additional ray from the point of intersection to each of the light sources. This is shown for a single light source in Fig.

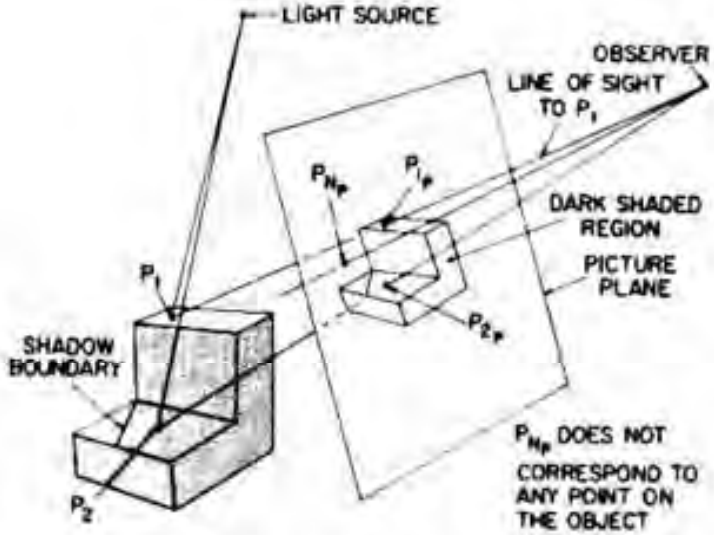


Fig. 16.51 Determining whether a point on an object is in shadow. (Courtesy of Arthur Appel, IBM T.J. Watson Research Center.)

16.51, which is reproduced from a paper by Appel [APPE68]—the first paper published on ray tracing for computer graphics. If one of these *shadow rays* intersects any object along the way, then the object is in shadow at that point and the shading algorithm ignores the contribution of the shadow ray's light source. Figure 16.52 shows two pictures that Appel rendered with this algorithm, using a pen plotter. He simulated a halftone pattern by placing a different size "+" at each pixel in the grid, depending on the pixel's intensity. To compensate for the grid's coarseness, he drew edges of visible surfaces and of shadows using a visible-line algorithm.

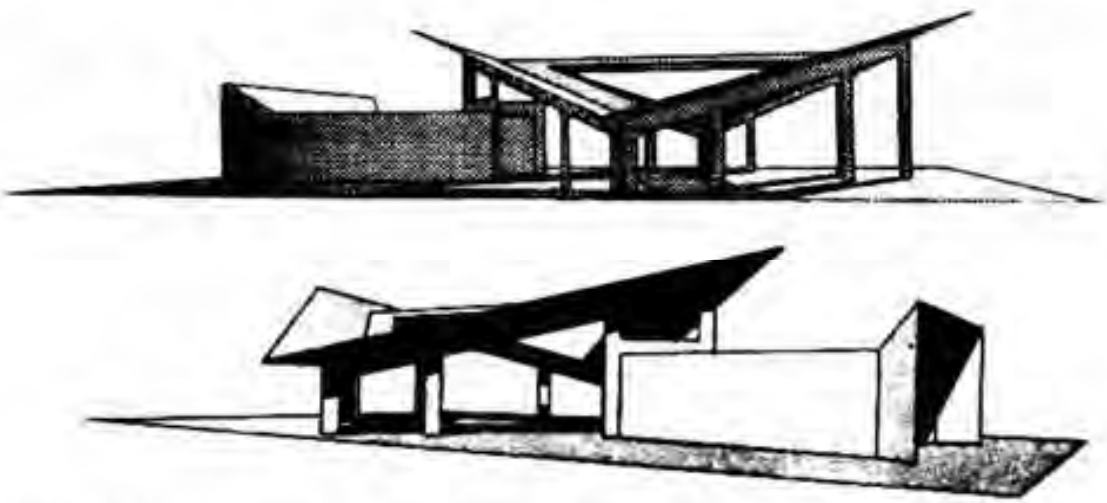


Fig. 16.52 Early pictures rendered with ray tracing. (Courtesy of Arthur Appel, IBM T.J. Watson Research Center.)

The illumination model developed by Whitted [WHIT80] and Kay [KAY79a] fundamentally extended ray tracing to include specular reflection and refractive transparency. Color Plate III.10 is an early picture generated with these effects. In addition to shadow rays, Whitted's recursive ray-tracing algorithm conditionally spawns *reflection rays* and *refraction rays* from the point of intersection, as shown in Fig. 16.53. The shadow, reflection, and refraction rays are often called *secondary rays*, to distinguish them from the *primary rays* from the eye. If the object is specularly reflective, then a reflection ray is reflected about the surface normal in the direction of $\bar{R}$, which may be computed as in Section 16.1.4. If the object is transparent, and if total internal reflection does not occur, then a refraction ray is sent into the object along $\bar{T}$ at an angle determined by Snell's law, as described in Section 16.5.2. (Note that your incident ray may be oppositely oriented to those in these sections.)

Each of these reflection and refraction rays may, in turn, recursively spawn shadow, reflection, and refraction rays, as shown in Fig. 16.54. The rays thus form a *ray tree*, such as that of Fig. 16.55. In Whitted's algorithm, a branch is terminated if the reflected and refracted rays fail to intersect an object, if some user-specified maximum depth is reached or if the system runs out of storage. The tree is evaluated bottom-up, and each node's intensity is computed as a function of its children's intensities. Color Plate III.11(a) and (b) were made with a recursive ray-tracing algorithm.

We can represent Whitted's illumination equation as

$$I_\lambda = I_{a\lambda} k_a O_{d\lambda} + \sum_{1 \le i \le m} S_i f_{att_i} I_{p\lambda_i} [k_d O_{d\lambda} (\bar{N} \cdot \bar{L}_i) + k_s (\bar{N} \cdot \bar{H}_i)^n] + k_r I_{r\lambda} + k_t I_{t\lambda}, \quad (16.55)$$

where $I_{r\lambda}$ is the intensity of the reflected ray, k_t is the *transmission coefficient* ranging between 0 and 1, and $I_{t\lambda}$ is the intensity of the refracted transmitted ray. Values for $I_{r\lambda}$ and $I_{t\lambda}$ are determined by recursively evaluating Eq. (16.55) at the closest surface that the reflected and transmitted rays intersect. To approximate attenuation with distance, Whitted multiplied the I_λ calculated for each ray by the inverse of the distance traveled by the ray. Rather than treating S_i as a delta function, as in Eq. (16.24), he also made it a continuous function

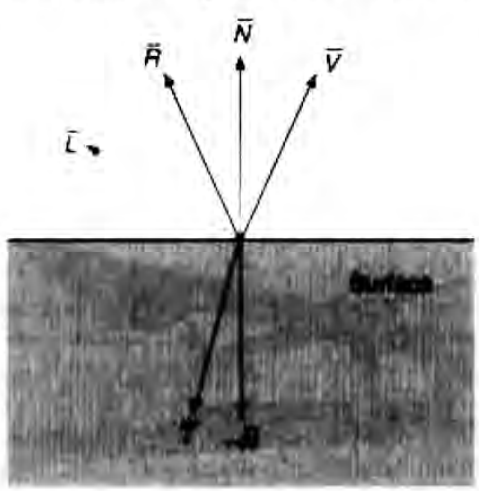


Fig. 16.53 Reflection, refraction, and shadow rays are spawned from a point of intersection

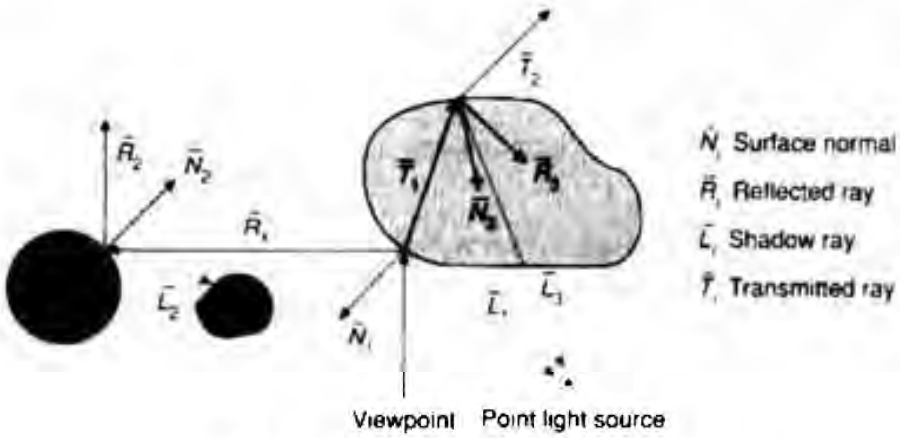


Fig. 16.54 Rays recursively spawn other rays.

of the k_s of the objects intersected by the shadow ray, so that a transparent object obscures less light than an opaque one at those points it shadows.

Figure 16.56 shows pseudocode for a simple recursive ray tracer. RT_trace determines the closest intersection the ray makes with an object and calls RT_shade to determine the shade at that point. First, RT_shade determines the intersection's ambient color. Next, a shadow ray is spawned to each light on the side of the surface being shaded to determine its contribution to the color. An opaque object blocks the light totally, whereas a transparent one scales the light's contribution. If we are not too deep in the ray tree, then recursive calls are made to RT_trace to handle reflection rays for reflective objects and refraction rays for transparent objects. Since the indices of refraction of two media are needed to determine the direction of the refraction ray, the index of refraction of the material in which a ray is traveling can be included with each ray. RT_trace retains the ray tree only long enough to determine the current pixel's color. If the ray trees for an entire image can be preserved, then surface properties can be altered and a new image recomputed relatively quickly, at the

Fig. 16.55 The ray tree for Fig. 16.54

```
select center of projection and window on view plane;
for (each scan line in image) {
    for (each pixel in scan line) {
        determine ray from center of projection through pixel;
        pixel = RT_trace (ray, 1);
    }
}

/* Intersect ray with objects and compute shade at closest intersection. */
/* Depth is current depth in ray tree. */

RT_color RT_trace (RT_ray ray, int depth)
{
    determine closest intersection of ray with an object;

    if (object hit) {
        compute normal at intersection;
        return RT_shade (closest object hit, ray, intersection, normal, depth);
    } else
        return BACKGROUND_VALUE;
}  /* RT_trace */

/* Compute shade at point on object, tracing rays for shadows, reflection and refraction. */
RT_color RT_shade (
    RT_object object,           /* Object intersected */
    RT_ray ray,                 /* Incident ray */
    RT_point point,             /* Point of intersection to shade */
    RT_normal normal,           /* Normal at point */
    int depth)                  /* Depth in ray tree */
{
    RT_color color;             /* Color of ray */
    RT_ray rRay, tRay, sRay;    /* Reflected, refracted, and shadow rays */
    RT_color rColor, tColor;    /* Reflected and refracted ray colors. */

    color = ambient term;
    for (each light) {
        sRay = ray to light from point;
        if (dot product of normal and direction to light is positive) {
            compute how much light is blocked by opaque and transparent surfaces,
                and use to scale diffuse and specular terms before adding them to color;
        }
    }
```

Fig. 16.56 *(Cont'd.)*

```
    if (depth < maxDepth) {        /* Return if depth is too deep. */
        if (object is reflective) {
            rRay = ray in reflection direction from point;
            rColor = RT_trace (rRay, depth + 1);
            scale rColor by specular coefficient and add to color;
        }

        if (object is transparent) {
            tRay = ray in refraction direction from point;
            if (total internal reflection does not occur) {
                tColor = RT_trace (tRay, depth + 1);
                scale tColor by transmission coefficient and add to color;
            }
        }
    }
    return color;                  /* Return color of ray */
} /* RT_shade */
```

Fig. 16.56 Pseudocode for simple recursive ray tracing without antialiasing.

cost of only reevaluating the trees. Sequin and Smyrl [SEQU89] present techniques that minimize the time and space needed to process and store ray trees.

Figure 16.54 shows a basic problem with how ray tracing models refraction: The shadow ray $\overline{L}_3$ is not refracted on its path to the light. In fact, if we were to simply refract $\overline{L}_3$ from its current direction at the point where it exits the large object, it would not end at the light source. In addition, when the paths of rays that are refracted are determined, a single index of refraction is used for each ray. Later, we discuss some ways to address these failings.

Ray tracing is particularly prone to problems caused by limited numerical precision. These show up when we compute the objects that intersect with the secondary rays. After the x, y, and z coordinates of the intersection point on an object visible to an eye ray have been computed, they are then used to define the starting point of the secondary ray for which we must determine the parameter t (Section 15.10.1). If the object that was just intersected is intersected with the new ray, it will often have a small, nonzero t, because of numerical-precision limitations. If not dealt with, this false intersection can result in visual problems. For example, if the ray were a shadow ray, then the object would be considered as blocking light from itself, resulting in splotchy pieces of incorrectly "self-shadowed" surface. A simple way to solve this problem for shadow rays is to treat as a special case the object from which a secondary ray is spawned, so that intersection tests are not performed on it. Of course, this does not work if objects are supported that really could obscure themselves or if transmitted rays have to pass through the object and be reflected from the inside of the same object. A more general solution is to compute abs(t) for an intersection, to compare it with a small tolerance value, and to ignore it if it is below the tolerance.

The paper Whitted presented at *SIGGRAPH '79* [WHIT80], and the movies he made using the algorithm described there, started a renaissance of interest in ray tracing. Recursive ray tracing makes possible a host of impressive effects—such as shadows, specular reflection, and refractive transparency—that were difficult or impossible to obtain previously. In addition, a simple ray tracer is quite easy to implement. Consequently, much effort has been directed toward improving both the algorithm's efficiency and its image quality. We provide a brief overview of these issues here, and discuss several parallel hardware implementations that take advantage of the algorithm's intrinsic parallelism in Section 18.11.2. For more detail, see [GLAS89].

16.12.1 Efficiency Considerations for Recursive Ray Tracing

Section 15.10.2 discussed how to use extents, hierarchies, and spatial partitioning to limit the number of ray–object intersections to be calculated. These general efficiency techniques are even more important here than in visible-surface ray tracing for several reasons. First, a quick glance at Fig. 16.55 reveals that the number of rays that must be processed can grow exponentially with the depth to which rays are traced. Since each ray may spawn a reflection ray and a refraction ray, in the worst case, the ray tree will be a complete binary tree with $2^n - 1$ rays, where the tree depth is n. In addition, each reflection or refraction ray that intersects with an object spawns one shadow ray for each of the m light sources. Thus, there are potentially $m(2^n - 1)$ shadow rays for each ray tree. To make matters worse, since rays can come from any direction, traditional efficiency ploys, such as clipping objects to the view volume and culling back-facing surfaces relative to the eye, cannot be used in recursive ray tracing. Objects that would otherwise be invisible, including back faces, may be reflected from or refracted through visible surfaces.

Item buffers. One way of speeding up ray tracing is simply not to use it at all when determining those objects directly visible to the eye. Weghorst, Hooper, and Greenberg [WEGH84] describe how to create an *item buffer* by applying a less costly visible-surface algorithm, such as the z-buffer algorithm, to the scene, using the same viewing specification. Instead of determining the shade at each pixel, however, they record in the item buffer pixel the identity of the closest object. Then, only this object needs to be processed by the ray tracer to determine the eye ray's exact intersection for this pixel, so that further rays may be spawned.

Reflection maps. Tracing rays can be avoided in other situations, too. Hall [HALL86] shows how to combine ray tracing with the reflection maps discussed in Section 16.6. The basic idea is to do less work for the secondary rays than for primary rays. Those objects that are not directly visible in an image are divided into two groups on the basis of an estimation of their indirect visibility. Ray tracing is used to determine the global lighting contributions of the more visible ones, whereas indexing into a suitably prepared reflection map handles the others. One way to estimate the extent to which an object is indirectly visible is to measure the solid angle subtended by the directly visible objects as seen from the centroid of the indirectly visible object. If the solid angle is greater than some threshold, then the object will be included in the environment to be traced by reflection rays (this environment

includes the directly visible objects); otherwise, the object will be represented only in the reflection map. When ray tracing is performed, if a reflection ray does not intersect one of the objects in the reflection-ray environment, then the ray is used to index into the reflection map. Hall also points out that reflected and refracted images are often extremely distorted. Therefore, good results may be achieved by intersecting the reflection and refraction rays with object definitions less detailed than those used for the eye rays.

Adaptive tree-depth control. Although ray tracing is often used to depict highly specular objects, most of an image's area is usually not filled with such objects. Consequently, a high recursion level often results in unnecessary processing for large parts of the picture. Hall [HALL83] introduced the use of *adaptive tree-depth control*, in which a ray is not cast if its contribution to the pixel's intensity is estimated to be below some preset threshold. This is accomplished by approximating a ray's maximum contribution by calculating its intensity with the assumption that the ray's child rays have intensities of 1. This allows the ray's contribution to its parent to be estimated. As the ray tree is built, the maximum contribution of a ray is multiplied by those of its ancestors to derive the ray's maximum contribution to the pixel. For example, suppose that $\overline{R}_1$ and $\overline{R}_2$ in Fig. 16.55 are spawned at surfaces with k_s values of .1 and .05, respectively. At the first intersection, we estimate the maximum contribution to the pixel of $\overline{R}_1$ to be .1. At the second intersection, we estimate the maximum contribution to the pixel of $\overline{R}_2$ to be $.05 \times .1 = .005$. If this is below our threshold, we may decide not to cast $\overline{R}_2$. Although adaptive tree-depth control has been shown to work well for many images, it is easy to design cases in which it will fail. Although one uncast ray may have an imperceptible effect on a pixel's shade, a pixel may receive a significant amount of light from a large number of individually insignificant rays.

Light buffers. We noted that m shadow rays are spawned for each reflection or refraction ray that hits an object. Shadow rays are special, however, in that each is fired toward one of a relatively small set of objects. Haines and Greenberg [HAIN86] have introduced the notion of a *light buffer* to increase the speed with which shadow rays are processed. A light buffer is a cube centered about a light source and aligned with the world-coordinate axes, as shown in Fig. 16.57(a). Each side is tiled with a regular grid of squares, and each square is associated with a depth-sorted list of surfaces that can be seen through it from the light. The

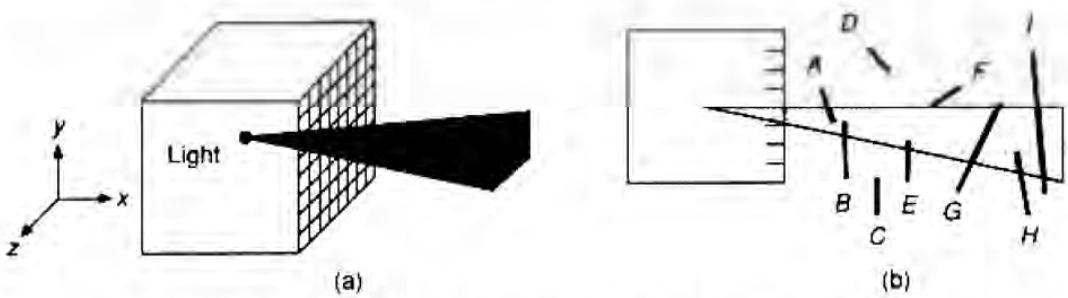


Fig. 16.57 Light buffer centered about a light source. (a) Volume defined by one square. (b) Cross-section shows volume of square in 2D with intersected surfaces. Square's depth-sorted list is initially *A, B, E, G, H,* and *I*.

lists are filled by scan converting the objects in the scene onto each face of the light buffer with the center of projection at the light. The scan-conversion algorithm places an object on the list of every square covered by that object's projection, no matter how small the overlap. Figure 16.57(b) shows a square's set of intersecting surfaces. A shadow ray is processed by determining the light-buffer square through which it passes. The ray needs to be tested only for intersection against that square's ordered list of surfaces. Thus, a light buffer implements a kind of 3D spatial partitioning of the 3D view of its light.

A number of efficiency speedups are possible. For example, any object whose projection onto the light buffer totally covers a square (e.g., G in Fig. 16.57b) can be associated with a special "full-occlusion" record in the square's list; any object more distant from the light than a fully occluding object is purged from the list. In this case, H and I are purged. Whenever the ray from an intersection point is tested against a square's list, the test can be terminated immediately if there is a full-occlusion record closer to the light than the intersection point. In addition, back-face culling may be used to avoid adding any faces to a list that are both part of a closed opaque solid and back facing relative to the light. Color Plate III.12 was made using the light-buffer technique.

Haines and Greenberg also mention an interesting use of object coherence to determine shadows that can be applied even without using light buffers. A pointer is associated with each light and is initialized to null. When an object is found to be shadowed from a light by some opaque surface, the light's pointer is set to the shadowing surface. The next time a shadow ray is cast for the light, it is first intersected with the object pointed at by the light's pointer. If the ray hits it, then intersection testing is finished; otherwise, the pointer is set to null and testing continues.

Ray classification. The spatial-partitioning approaches discussed in Section 15.10.2 make it possible to determine which objects lie in a given region of 3D space. Arvo and Kirk [ARVO87] have extended this concept to partition rays by the objects that they intersect, a technique called *ray classification*. A ray may be specified by its position in 5D *ray space*, determined by the 3D position of its origin in space and its 2D direction in spherical coordinates. A point in ray space defines a single ray, whereas a finite subset of ray space defines a family of rays or *beam*. Ray classification adaptively partitions ray space into subsets, each of which is associated with a list of objects that it contains (i.e., that one of its rays could intersect). To determine the candidate objects that may be intersected by any ray, we need only to retrieve the list of objects associated with the subset of ray space in which the ray resides.

Instead of using spherical coordinates to specify direction, Arvo and Kirk use the (u, v) coordinate system on each of the six sides of an axis-aligned cube centered about a ray's origin. They subdivide ray space with six copies of a 5D bintree (Section 12.6.3); one copy is needed for each direction along an axis, since a (u, v) pair specifies a position on the cube's side, but not the side itself. As shown in Fig. 16.58(a), in 2D a set of ray origins (a rectangle in (x, y)), combined with a set of ray directions (an interval in u on one of four sides), define a partially bounded polygon that defines the beam. In 3D, shown in Fig. 16.58(b), a set of ray origins (a rectangular parallepiped in (x, y, z)), combined with a set of ray directions (a rectangle in (u, v)), defines a partially bounded polyhedral volume that defines the beam. Objects (or their extents) may be intersected with this volume (e.g., using

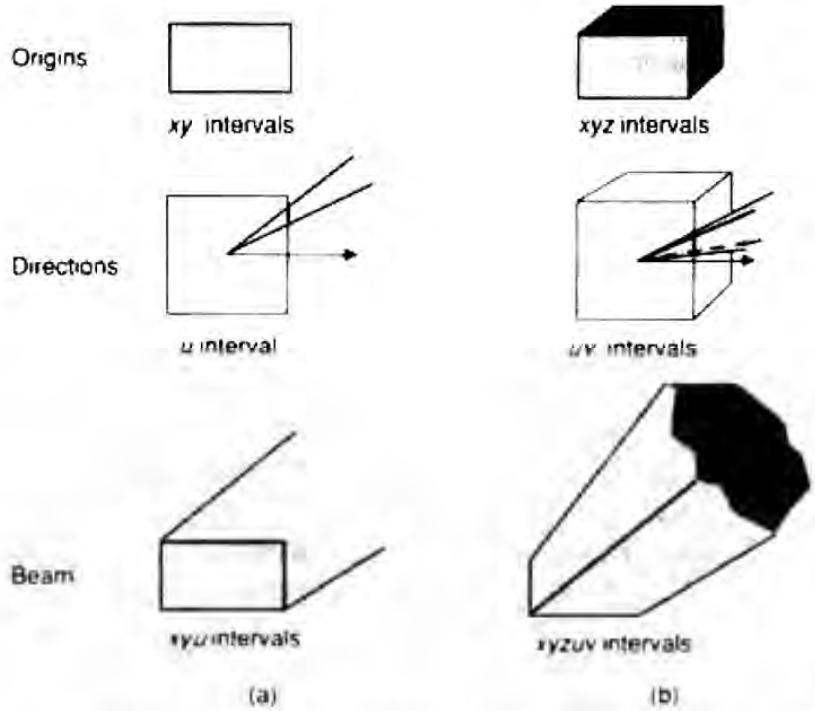


Fig. 16.58 Ray classification: (a) 2-space beams; (b) 3-space beams. (After [ARVO87].)

the BSP-tree-based approach of Section 12.6.4) to determine whether they are contained in it. Initially, each unpartitioned bintree represents positions in all of the environment's 3-space extent, and directions that pass through any part of the bintree's designated side. Each bintree is initially associated with all the objects in the environment.

Arvo and Kirk use *lazy evaluation* to build the trees at the same time that they trace rays. When a ray is traced, its direction selects the root of the bintree whose (u, v) coordinate system it will intersect. If more than some threshold number of objects exist at the node, the bintree is subdivided equally along each of its five axes. With each subdivision, only the child node in which the ray resides is processed; it inherits from its parent those objects that lie within the child. When subdivision stops, only those objects associated with the ray's partition are returned for intersection. Partitioning data structures as the rays are traced minimizes the amount of processing done for objects that are not intersected.

16.12.2 A Better Illumination Model

It is possible to make a number of extensions to Eq. (16.55). Hall [HALL83] has developed a model in which the specular light expressions are scaled by a wavelength-dependent Fresnel reflectivity term. An additional term for transmitted local light is added to take into account the contribution of transmitted light directly emitted by the light sources, and is also scaled by the Fresnel transmissivity term. The global reflected and refracted rays

further take into account the transmittance of the medium through which they travel. Hall's model may be expressed as

$$I_\lambda = I_{a\lambda} k_a O_{d\lambda} + \sum_{1 \le i \le m} S_i f_{att_i} I_{p\lambda_i} [k_d O_{d\lambda}(\overline{N} \cdot \overline{L}_i) + k_s(\overline{N} \cdot \overline{H}_i)^n F_\lambda + k_s(\overline{N} \cdot \overline{H'}_i)^n T_\lambda]$$
$$+ k_s F_\lambda I_{r\lambda} A_r^{d_r} + k_s T_\lambda I_{t\lambda} A_t^{d_t}, \tag{16.56}$$

where d_r is the distance traveled by the reflected ray, d_t is the distance traveled by the transmitted ray, A_r is the transmissivity per unit length of material for the reflected ray, and A_t is the transmissivity per unit length of material for the transmitted ray. $\overline{H'}$ is the normal for those microfacets that are aligned such that they will refract light directly from the light source to the viewer. It may be computed as

$$\overline{H'} = \frac{\overline{V} - (\eta_{avg\theta_t} / \eta_{avg\theta_i})\overline{L}}{(\eta_{avg\theta_t} / \eta_{avg\theta_i}) - 1}, \tag{16.57}$$

where $\eta_{avg\theta_t}$ is the average coefficient of refraction of the object through which the ray to the light passes. T_λ is the Fresnel transmissivity term for the material. Note that k_t is not used here; instead, k_s is scaled by either F_λ or T_λ. Since all light that is not reflected is transmitted (and possibly) absorbed, $F_\lambda + T_\lambda = 1$.

Color Plate III.13 shows the same scene rendered with Whitted's model and with Hall's model. The color of the glass sphere in part (b) clearly shows the filtering effects resulting from the use of the transmissivity terms.

16.12.3 Area-Sampling Variations

One of conventional ray tracing's biggest drawbacks is that this technique point samples on a regular grid. Whitted [WHIT80] suggested that unweighted area sampling could be accomplished by replacing each linear eye ray with a pyramid defined by the eye and the four corners of a pixel. These pyramids would be intersected with the objects in the environment, and sections of them would be recursively refracted and reflected by the objects that they intersect. A pure implementation of this proposal would be exceedingly complex, however, since it would have to calculate exact intersections with occluding objects, and to determine how the resulting pyramid fragments are modified as they are recursively reflected from and refracted by curved surfaces. Nevertheless, it has inspired several interesting algorithms that accomplish antialiasing, and at the same time decrease rendering time by taking advantage of coherence.

Cone tracing. *Cone tracing,* developed by Amanatides [AMAN84], generalizes the linear rays into cones. One cone is fired from the eye through each pixel, with an angle wide enough to encompass the pixel. The cone is intersected with objects in its path by calculating approximate fractional blockage values for a small set of those objects closest to the cone's origin. Refraction and reflection cones are determined from the optical laws of spherical mirrors and lenses as a function of the surface curvature of the object intersected and the area of intersection. The effects of scattering on reflection and refraction are simulated by further broadening the angles of the new reflection and refraction cones. The

soft-edged shadows of extended light sources are reproduced by modeling the sources as spheres. A shadow cone is generated whose base is the cross-section of the light source. The light source's intensity is then scaled by the fraction of the cone that is unblocked by intervening objects. Color Plate III.14 was rendered using cone tracing; the three spheres have successively rougher surfaces, and all cast soft shadows.

Beam tracing. *Beam tracing*, introduced by Heckbert and Hanrahan [HECK84], is an object-precision algorithm for polygonal environments that traces pyramidal beams, rather than linear rays. Instead of tracing beams through each pixel, as Whitted suggested, Heckbert and Hanrahan take advantage of coherence by beginning with a single beam defined by the viewing pyramid. The viewing pyramid's beam is intersected with each polygon in the environment, in front-to-back sorted order, relative to the pyramid's apex. If a polygon is intersected, and therefore visible, it must be subtracted from the pyramid using an algorithm such as that described in Section 19.1.4. For each visible polygon fragment, two polyhedral pyramids are spawned, one each for reflection and refraction. The algorithm proceeds recursively, with termination criteria similar to those used in ray tracing. The environment is transformed into each new beam's coordinate system by means of an appropriate transformation. Although beam tracing models reflection correctly, refraction is not a linear transformation since it bends straight lines, and the refracted beam is thus only approximated. Beam tracing produces an object-precision beam tree of polygons that may be recursively rendered using a polygon scan-conversion algorithm. Each polygon is rendered using a local illumination equation, and then its reflected and refracted child polygons are rendered on top and are averaged with it, taking into account the parent's specular and transmissive properties. Beam tracing takes advantage of coherence to provide impressive speedups over conventional ray tracing at the expense of a more complex algorithm, limited object geometry, and incorrectly modeled refraction. Color Plate III.15 was rendered using this algorithm.

Beam tracing can accommodate shadows by using a variant of the Atherton–Weiler–Greenberg shadow algorithm (Section 16.4.2). Beams are traced from the point of view of each light source to determine all surfaces directly visible from the light sources, and the resulting polygons are added to the data structure as lit detail polygons that affect only the shading. This produces shadows similar to those obtained with conventional ray tracing.

Pencil tracing. Shinya, Takahashi, and Naito [SHIN87] have implemented an approach called *pencil tracing* that solves some of the problems of cone tracing and beam tracing. A *pencil* is a bundle of rays consisting of a central *axial* ray, surrounded by a set of nearby *paraxial* rays. Each paraxial ray is represented by a 4D vector that represents its relationship to the axial ray. Two dimensions express the paraxial ray's intersection with a plane perpendicular to the axial ray; the other two dimensions express the paraxial ray's direction. In many cases, only an axial ray and solid angle suffice to represent a pencil. If pencils of sufficiently small solid angle are used, then reflection and refraction can be approximated well by a linear transformation expressed as a 4×4 matrix. Shinya, Takahashi, and Naito have developed error-estimation techniques for determining an appropriate solid angle for a pencil. Conventional rays must be traced where a pencil would intersect the edge of an object, however, since the paraxial transformations are not valid in these cases.

16.12.4 Distributed Ray Tracing

The approaches we have just discussed avoid the aliasing problems of regular point sampling by casting solid beams rather than infinitesimal rays. In contrast, *distributed ray tracing*, developed by Cook, Porter, and Carpenter [COOK84b], is based on a stochastic approach to supersampling that trades off the objectionable artifacts of aliasing for the less offensive artifacts of noise [COOK86]. As we shall see, the ability to perform antialiased spatial sampling can also be exploited to sample a variety of other aspects of the scene and its objects to produce effects such as motion blur, depth of field, extended light sources, and specular reflection from rough surfaces. The word *distributed* in this technique's name refers to the fact that rays are stochastically distributed to sample the quantities that produce these effects. The basic concepts have also been applied to other algorithms besides ray tracing [COOK87].

Stochastic sampling. As explained in Section 14.10, aliasing results when a signal is sampled with regularly spaced samples below the Nyquist rate. This is true even if we supersample and filter to compute the value of a pixel. If the samples are not regularly spaced, however, the sharply defined frequency spectrum of the aliases is replaced by noise, an artifact that viewers find much less objectionable than the individually recognizable frequency components of regular aliasing, such as staircasing.

It is not enough, however, merely to replace ray tracing's regular grid of eye rays with an equal number of rays passing through random points on the image plane, since purely random samples cluster together in some areas and leave others unsampled. Cook [COOK86] suggests the desirability of a minimum-distance Poisson distribution in which no pair of random samples is closer than some minimum distance. Calculating such a distribution is expensive, however, and even if one is created in advance, along with filters to determine each sample's contributions to neighboring pixels, a very large look-up table will be required to store the information. Instead, a satisfactory approximation to the minimum-distance Poisson distribution is obtained by displacing by a small random distance the position of each element of a regularly spaced sample grid. This technique is called *jittering*. In sampling the 2D image plane, each sample in a regular grid is jittered by two uncorrelated random quantities, one each for x and y, both generated with a sufficiently small variance that the samples do not overlap (Fig. 16.59). Figure 16.60 shows a minimum-distance Poisson distribution and a jittered regular distribution. In fact, if the

Fig. 16.59 Jittered sampling. Each sample in a regular 2D grid is jittered by two small uncorrelated random quantities. × = original position; ● = position after jitter.

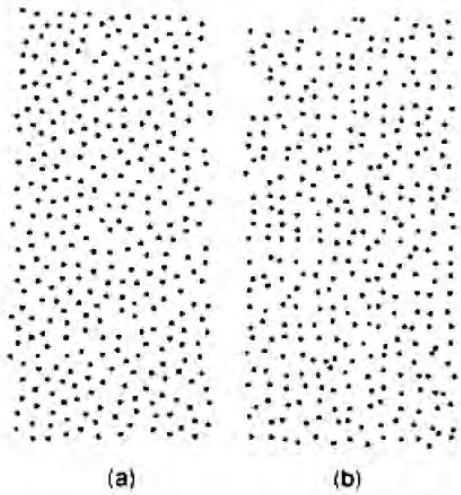


Fig. 16.60 (a) A minimum distance Poisson distribution. (b) A jittered regular distribution. (Courtesy of Mark A. Z. Dippé and Earl Wold, University of California, Berkeley.)

amount of jitter is small compared to the filter width, then the filter can be precomputed, taking into account the positions of the unjittered samples. Cook, Porter, and Carpenter found that a 4 × 4 subpixel grid is adequate for most situations. Poisson and jittered sampling are analyzed in [DIPP85], and strategies for performing adaptive stochastic sampling, including the statistical analysis of samples to determine whether to place new ones, are discussed in [DIPP85; LEE85b; KAJI86; MITC87].

Figure 16.61 compares the use of regularly spaced samples with and without added jitter to sample frequencies at rates above and below the Nyquist rate. In Fig. 16.61(a), sampling above the Nyquist rate, the shape of the signal is captured well, but with some added noise. In Fig. 16.61(b), the sampled amplitude is totally random if there is an integral number of cycles in the sampled range. If there is a fractional number of cycles in the range, then some parts of the waveform have a better chance of being sampled than do others, and thus a combination of aliasing and noise will result. The higher the frequency, the greater the proportion of noise to aliasing. Figure 16.62 demonstrates how a comb of regularly spaced triangles, each $(n + 1)/n$ pixels wide, produces an aliased image when it is sampled by regularly spaced sample points, and produces a noisy image when the sample points are jittered.

Sampling other dimensions. As long as the extra rays needed for spatial antialiasing have been cast, this same basic technique of stochastic sampling can also be used to distribute the rays to sample other aspects of the environment. Motion blur is produced by distributing rays in time. Depth of field is modeled by distributing the rays over the area of the camera lens. The blurred specular reflections and translucent refraction of rough surfaces are simulated by distributing the rays according to the specular reflection and transmission functions. Soft shadows are obtained by distributing the shadow rays over the solid angle subtended by an extended light source as seen from the point being shaded. In all cases, distributed ray tracing uses stochastic sampling to perturb the same rays that would be cast to accomplish spatial antialiasing alone.

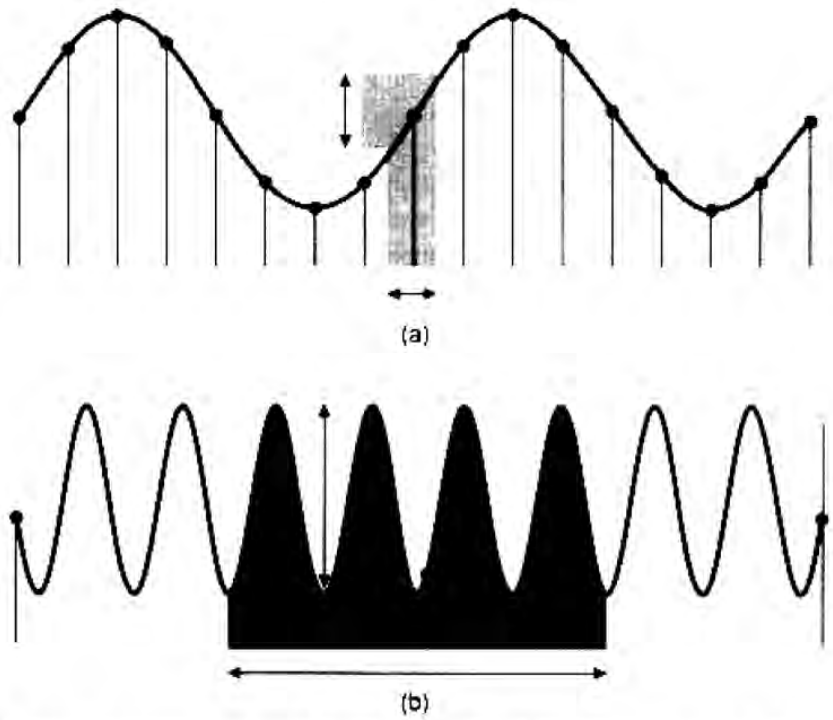


Fig. 16.61 Use of regularly spaced samples with added jitter to sample frequencies (a) above and (b) below the Nyquist rate. The nominal position of each sample is shown as a dot (•). Horizontal arrows indicate the range of jittered sample positions. Vertical arrows indicate how much sampled values can vary. (After [COOK86].)

Sampling in a nonspatial dimension is accomplished by associating each of the pixel's subsampled rays with a range of the value being sampled. Jittering is then used to determine the exact sample point. The ranges may be allocated by partitioning the entire interval being sampled into the same number of subintervals as there are subpixels and randomly allocating subintervals to subpixels. Thus, subpixel ij of each pixel is always associated with the same range for a particular dimension. It is important to ensure that the method of allocating the ranges for each dimension does not correlate the values of any two dimensions. For example, if temporally earlier samples tended to sample the left side of an extended light source, and later samples tended to sample the right side, then obscuring the right side of the light early in the temporal interval being depicted might have no effect on the shadow cast. In the case of temporal sampling, each object being intersected must first be moved to its position at the point in time associated with the sampling ray. Cook [COOK86] suggests computing a bounding box for the object's entire path of motion, so that the bounding-box test can be performed without the expense of moving the object.

A weighted distribution in some dimension can be simulated by applying unequal weights to evenly distributed samples. Figure 16.63(a) shows such a distribution. A more attractive alternative, however, is to use *importance sampling*, in which proportionately more sample points are located at positions of higher weight. This is accomplished by dividing the weighting filter into regions of equal area and assigning the same number of equally weighted sample points to each, as shown in Fig. 16.63(b). The amount of jitter

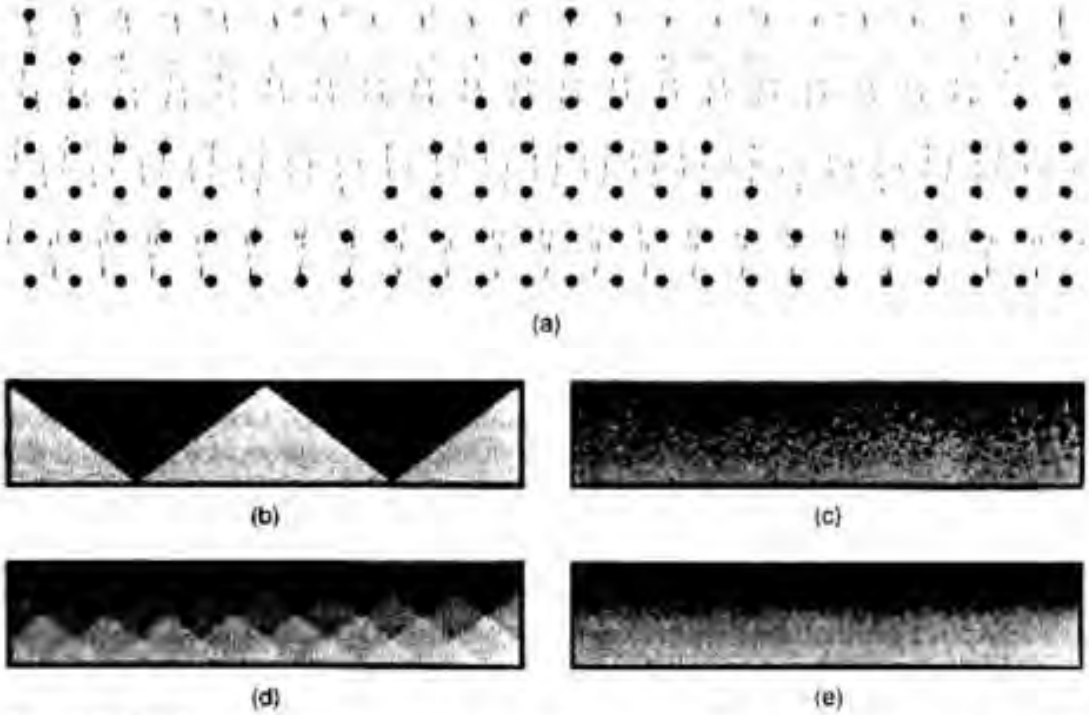


Fig. 16.62 Aliasing vs. noise. (a) A comb with regularly spaced triangles, each $(n + 1)/n$ pixels wide, sampled with one sample per pixel. o = samples that fall outside comb; • = samples that fall inside comb. (b) A comb with 200 triangles, each 1.01 pixels wide and 50 pixels high. 1 sample/pixel, regular grid. (c) 1 sample/pixel, jittered $\pm\frac{1}{2}$ pixel. (d) 16 samples/pixel, regular grid. (e) 16 samples/pixel, jittered $\pm\frac{1}{8}$ pixel. (Images (b)–(e) by Robert Cook, Lucasfilm Ltd.)

associated with a region also varies in proportion to its size. Color Plate III.16 was created using distributed ray tracing. It shows five billiard balls with motion blur and penumbrae cast by extended light sources. Note the blurred shadows and reflections of the four moving billiard balls.

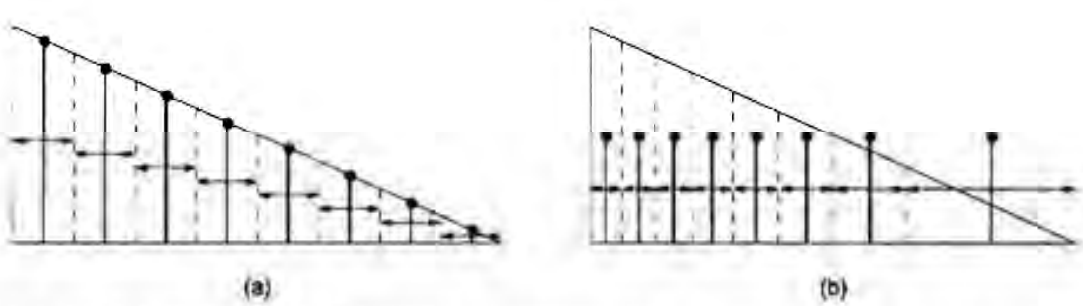


Fig. 16.63 Importance sampling is accomplished by partitioning the weighting function into regions of equal area. The horizontal axis is the dimension sampled; the vertical axis is the weighting. Dots show nominal position of samples; arrows show jitter range. (a) Evenly distributed, unequally weighted samples. (b) Importance sampling: unevenly distributed, equally weighted samples.

Path tracing and the integral equation method. Kajiya [KAJI86] has implemented an efficient variation on distributed ray tracing called *path tracing*. Rather than each ray being grown into a binary tree, exactly one reflection or refraction ray is fired at each intersection to form a linear path, along with one ray to each light source. The decision to shoot either a reflection or a refraction ray is guided by the desired distribution of the different kinds of rays for each pixel. Kajiya has also extended this algorithm to develop a solution to the rendering equation (Eq. 16.54), called the *integral equation method*, that takes into account all ways in which light can reach a point. He uses variance-reduction methods to calculate a random variable based on the specular, diffuse, and transmission coefficients at each intersection. The random variable is used to determine whether the single ray cast from the intersection will be a specular reflection, diffuse reflection, or refraction ray, and the ray's direction is then chosen by sampling. In addition, a shadow ray is cast to a point on a light source, also chosen using variance-reduction methods. Because diffuse rays are traced, this approach models diffuse interobject reflections, an effect that we shall discuss in Section 16.13. Color Plate III.17 was rendered with the integral equation method. All objects shown are gray, except for the floor and green glass balls. The gray objects reflect green light focused by the balls and reddish light diffusely reflected from the floor, phenomena not modeled by conventional ray tracing (or by path tracing).

Kajiya's two approaches evaluate the diffuse reflection separately at each pixel, even though diffuse reflection tends to change relatively little from one pixel to the next. Ward, Rubinstein, and Clear [WARD88] have supplemented a ray tracer with a recursive diffuse reflection stage in which rays are used to trace some number of diffuse bounces from a surface to others illuminating it. Rather than computing the diffuse reflection for each pixel separately, they instead cache all of the values computed. When a pixel is processed, they use an estimate of the illuminance gradient at each cached diffuse reflection value "nearby" the pixel's intersection point to estimate the error associated with using that value. If the error is considered acceptable, then a weighted average of these cached values is used to compute a new value for the pixel; otherwise, a new diffuse calculation is made by tracing rays that sample the hemisphere, and its value is cached. The cached values can then be reused in computing other views of the same scene. Color Plate III.18 shows a series of images rendered using this technique, with different numbers of diffuse bounces. Color Plate III.19 contrasts a photograph of a conference room with a rendered image.

16.12.5 Ray Tracing from the Light Sources

One serious problem with ray tracing is caused by tracing all rays from the eye. Shadow rays are cast only to direct sources of light that are treated separately by the algorithm. Therefore, the effects of indirect reflected and refracted light sources, such as mirrors and lenses, are not reproduced properly: Light rays bouncing off a mirror do not cast shadows, and the shadows of transparent objects do not evidence refraction, since shadow rays are cast in a straight line toward the light source.

It might seem that we would need only to run a conventional ray tracer "backward" from the light sources to the eye to achieve these effects. This concept has been called *backward ray tracing*, to indicate that it runs in the reverse direction from regular ray tracing, but it is also known as *forward ray tracing* to stress that it follows the actual path from the lights to the eye. We call it *ray tracing from the light sources* to avoid confusion!

Done naively, ray tracing from the light sources results in new problems, since an insufficient number of rays ever would strike the image plane, let alone pass through the focusing lens or pinhole. Instead, ray tracing from the light sources can be used to supplement the lighting information obtained by regular ray tracing. Heckbert and Hanrahan [HECK84] suggest an elaboration of their proposed beam-tracing shadow method (Section 16.12.3) to accomplish this. If a light's beam tree is traced recursively, successive levels of the tree below the first level represent indirectly illuminated polygon fragments. Adding these to the database as surface-detail polygons allows indirect specular illumination to be modeled.

Arvo [ARVO86] has implemented a ray tracer that uses a preprocessing step in which rays from each light source are sent into the environment. Each ray is assigned an initial quota of energy, some of which is deposited at each intersection it makes with a diffusely reflecting object. He compensates for the relative sparseness of ray intersections by mapping each surface to a regular rectangular grid of counters that accumulate the deposited energy. Each ray's contribution is bilinearly partitioned among the four counters that bound the grid box in which the ray hits. A conventional ray-tracing pass is then made, in which the first pass's interpolated contributions at each intersection are used, along with the intensities of the visible light sources, to compute the diffuse reflection. Unfortunately, if a light ray strikes an object on the invisible side of a silhouette edge as seen from the eye, the ray can affect the shading on the visible side. Note that both these approaches to ray tracing from the light sources use purely specular reflectivity geometry to propagate rays in both directions.

16.13 RADIOSITY METHODS

Although ray tracing does an excellent job of modeling specular reflection and dispersionless refractive transparency, it still makes use of a directionless ambient-lighting term to account for all other global lighting contributions. Approaches based on thermal-engineering models for the emission and reflection of radiation eliminate the need for the ambient-lighting term by providing a more accurate treatment of interobject reflections. First introduced by Goral, Torrance, Greenberg, and Battaile [GORA84] and by Nishita and Nakamae [NISH85a], these algorithms assume the conservation of light energy in a closed environment. All energy emitted or reflected by every surface is accounted for by its reflection from or absorption by other surfaces. The rate at which energy leaves a surface, called its *radiosity*, is the sum of the rates at which the surface emits energy and reflects or transmits it from that surface or other surfaces. Consequently, approaches that compute the radiosities of the surfaces in an environment have been named *radiosity methods*. Unlike conventional rendering algorithms, radiosity methods first determine all the light interactions in an environment in a view-independent way. Then, one or more views are rendered, with only the overhead of visible-surface determination and interpolative shading.

16.13.1 The Radiosity Equation

In the shading algorithms considered previously, light sources have always been treated separately from the surfaces they illuminate. In contrast, radiosity methods allow any surface to emit light; thus, all light sources are modeled inherently as having area. Imagine

breaking up the environment into a finite number n of discrete patches, each of which is assumed to be of finite size, emitting and reflecting light uniformly over its entire area. If we consider each patch to be an opaque Lambertian diffuse emitter and reflector, then, for surface i,

$$B_i = E_i + \rho_i \sum_{1 \le j \le n} B_j F_{j-i} \frac{A_j}{A_i}. \tag{16.58}$$

B_i and B_j are the radiosities of patches i and j, measured in energy/unit time/unit area (i.e., W / m^2). E_i is the rate at which light is emitted from patch i and has the same units as radiosity. ρ_i is patch i's reflectivity and is dimensionless. F_{j-i} is the dimensionless *form factor* or *configuration factor*, which specifies the fraction of energy leaving the entirety of patch j that arrives at the entirety of patch i, taking into account the shape and relative orientation of both patches and the presence of any obstructing patches. A_i and A_j are the areas of patches i and j.

Equation (16.58) states that the energy leaving a unit area of surface is the sum of the light emitted plus the light reflected. The reflected light is computed by scaling the sum of the incident light by the reflectivity. The incident light is in turn the sum of the light leaving the entirety of each patch in the environment scaled by the fraction of that light reaching a unit area of the receiving patch. $B_j F_{j-i}$ is the amount of light leaving a unit area of A_j that reaches all of A_i. Therefore, it is necessary to multiply by the area ratio A_j / A_i to determine the light leaving all of A_j that reaches a unit area of A_i.

Conveniently, a simple reciprocity relationship holds between form factors in diffuse environments,

$$A_i F_{i-j} = A_j F_{j-i}. \tag{16.59}$$

Thus, Eq. (16.58) can be simplified, yielding

$$B_i = E_i + \rho_i \sum_{1 \le j \le n} B_j F_{i-j}. \tag{16.60}$$

Rearranging terms,

$$B_i - \rho_i \sum_{1 \le j \le n} B_j F_{i-j} = E_i. \tag{16.61}$$

Therefore, the interaction of light among the patches in the environment can be stated as a set of simultaneous equations:

$$\begin{bmatrix} 1 - \rho_1 F_{1-1} & -\rho_1 F_{1-2} & \cdots & -\rho_1 F_{1-n} \\ -\rho_2 F_{2-1} & 1 - \rho_2 F_{2-2} & \cdots & -\rho_2 F_{2-n} \\ \vdots & \vdots & \cdots & \vdots \\ \vdots & \vdots & \cdots & \vdots \\ -\rho_n F_{n-1} & -\rho_n F_{n-2} & \cdots & 1 - \rho_n F_{n-n} \end{bmatrix} \begin{bmatrix} B_1 \\ B_2 \\ \vdots \\ \vdots \\ B_n \end{bmatrix} = \begin{bmatrix} E_1 \\ E_2 \\ \vdots \\ \vdots \\ E_n \end{bmatrix}. \tag{16.62}$$

Note that a patch's contribution to its own reflected energy must be taken into account (e.g., it may be concave); so, in the general case, each term along the diagonal is not merely 1. Equation (16.62) must be solved for each band of wavelengths considered in the lighting model, since ρ_i and E_i are wavelength-dependent. The form factors, however, are

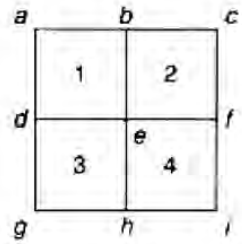


Fig. 16.64 Computing vertex radiosities from patch radiosities.

independent of wavelength and are solely a function of geometry, and thus do not need to be recomputed if the lighting or surface reflectivity changes.

Equation (16.62) may be solved using Gauss–Seidel iteration [PRES88], yielding a radiosity for each patch. The patches can then be rendered from any desired viewpoint with a conventional visible-surface algorithm; the set of radiosities computed for the wavelength bands of each patch are that patch's intensities. Instead of using faceted shading, we can compute vertex radiosities from the patch radiosities to allow intensity interpolation shading.

Cohen and Greenberg [COHE85] suggest the following approach for determining vertex radiosities. If a vertex is interior to a surface, it is assigned the average of the radiosities of the patches that share it. If it is on the edge, then the nearest interior vertex v is found. The radiosity of the edge vertex when averaged with B_v should be the average of the radiosities of the patches that share the edge vertex. Consider the patches in Fig. 16.64. The radiosity for interior vertex e is $B_e = (B_1 + B_2 + B_3 + B_4) / 4$. The radiosity for edge vertex b is computed by finding its nearest interior vertex, e, and noting that b is shared by patches 1 and 2. Thus, to determine B_b, we use the preceding definition; $(B_b + B_e) / 2 = (B_1 + B_2) / 2$. Solving for B_b, we get $B_b = B_1 + B_2 - B_e$. The interior vertex closest to a is also e, and a is part of patch 1 alone. Thus, since $(B_a + B_e) / 2 = B_1$, we get $B_a = 2B_1 - B_e$. Radiosities for the other vertices are computed similarly.

The first radiosity method was implemented by Goral et al. [GORA84], who used contour integrals to compute exact form factors for convex environments with no occluded surfaces, as shown in Color Plate III.20. Note the correct "color-bleeding" effects due to diffuse reflection between adjacent surfaces, visible in both the model and the rendered image: diffuse surfaces are tinged with the colors of other diffuse surfaces that they reflect. For radiosity methods to become practical, however, ways to compute form factors between occluded surfaces had first to be developed.

16.13.2 Computing Form Factors

Cohen and Greenberg [COHE85] adapted an image-precision visible-surface algorithm to approximate form factors for occluded surfaces efficiently. Consider the two patches shown in Fig. 16.65. The form factor from differential area dA_i to differential area dA_j is

$$dF_{di \to dj} = \frac{\cos\theta_i \cos\theta_j}{\pi r^2} H_{ij}\, dA_j \tag{16.63}$$

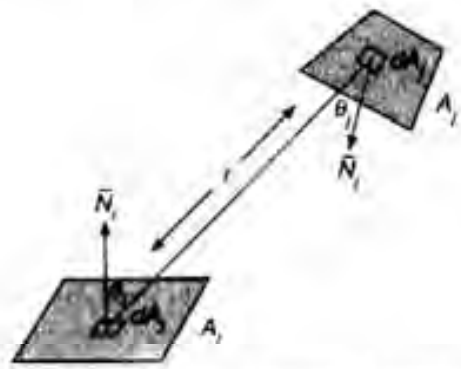


Fig. 16.65 Computing the form factor between a patch and a differential area.

For the ray between differential areas dA_i and dA_j in Fig. 16.65, θ_i is the angle that the ray makes with A_i's normal, θ_j is the angle that it makes with A_j's normal, and r is the ray's length. H_{ij} is either 1 or 0, depending on whether or not dA_j is visible from dA_i. To determine F_{di-j}, the form factor from differential area dA_i to finite area A_j, we need to integrate over the area of patch j. Thus,

$$F_{di-j} = \int_{A_j} \frac{\cos\theta_i \cos\theta_j}{\pi r^2} H_{ij}\, dA_j. \tag{16.64}$$

Finally, the form factor from A_i to A_j is the area average of Eq. (16.64) over patch i:

$$F_{i-j} = \frac{1}{A_i} \int_{A_i}\int_{A_j} \frac{\cos\theta_i \cos\theta_j}{\pi r^2} H_{ij}\, dA_i\, dA_j. \tag{16.65}$$

If we assume that the center point on a patch typifies the patch's other points, then F_{i-j} can be approximated by F_{di-j} computed for dA_i at patch i's center.

Nusselt has shown [SIEG81] that computing F_{di-j} is equivalent to projecting those parts of A_j that are visible from dA_i onto a unit hemisphere centered about dA_i, projecting this projected area orthographically down onto the hemisphere's unit circle base, and dividing by the area of the circle (Fig. 16.66). Projecting onto the unit hemisphere accounts for $\cos\theta_j / r^2$ in Eq. (16.64), projecting down onto the base corresponds to a multiplication by $\cos\theta_i$, and dividing by the area of the unit circle accounts for the π in the denominator.

Rather than analytically projecting each A_j onto a hemisphere, Cohen and Greenberg developed an efficient image-precision algorithm that projects onto the upper half of a cube centered about dA_i, with the cube's top parallel to the surface (Fig. 16.67). Each face of this *hemicube* is divided into a number of equal-sized square cells. (Resolutions used in pictures included in this book range from 50 by 50 to several hundred on a face.) All the other patches are clipped to the view-volume frusta defined by the center of the cube and each of its upper five faces, and then each of the clipped patches is projected onto the appropriate face of the hemicube. An item-buffer algorithm (Section 16.12.1) is used that records the

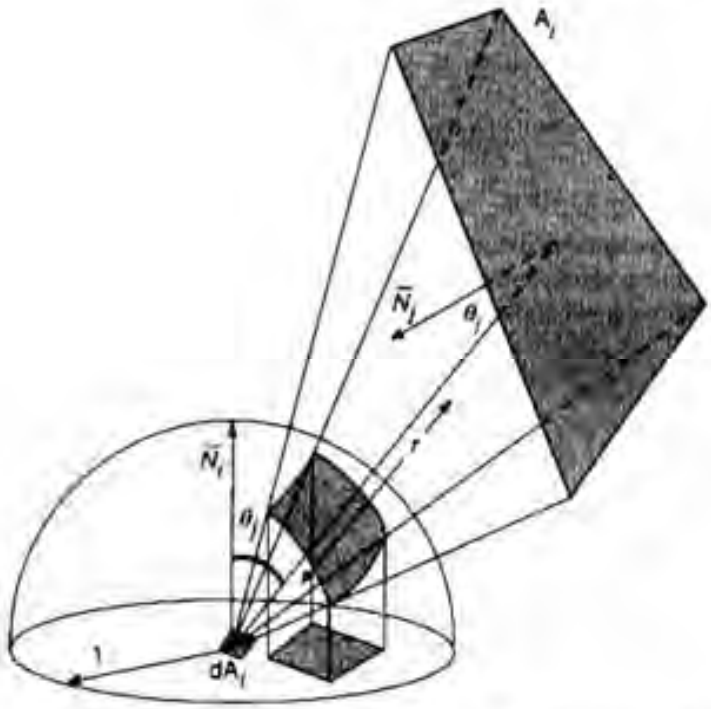


Fig. 16.66 Determining the form factor between a differential area and a patch using Nusselt's method. The ratio of the area projected onto the hemisphere's base to the area of the entire base is the form factor. (After [SIEG81].)

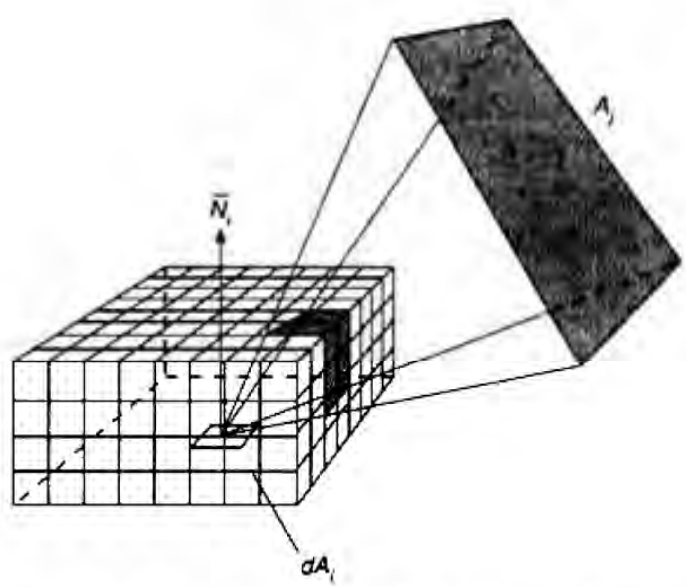


Fig. 16.67 The hemicube is the upper half of a cube centered about the patch. (After [COHE85].)

identity of the closest intersecting patch at each cell. Each hemicube cell p is associated with a precomputed delta form factor value,

$$\Delta F_p = \frac{\cos\theta_i \cos\theta_p}{\pi r^2}\Delta A, \tag{16.66}$$

where θ_p is the angle between the cell p's surface normal and the vector between dA_i and p, r is this vector's length, and ΔA is the area of a cell, as shown in Fig. 16.68. Assume that the hemicube has its own (x, y, z) coordinate system, with the origin at the center of the bottom face. For the top face in Fig. 16.68(a), we have

$$r = \sqrt{x_p^2 + y_p^2 + 1}, \tag{16.67}$$

$$\cos\theta_i = \cos\theta_p = \frac{1}{r},$$

where x_p and y_p are the coordinates of a hemicube cell. Thus, for the top face, Eq. (16.66) simplifies to

$$\Delta F_p = \frac{1}{\pi(x_p^2 + y_p^2 + 1)^2}\Delta A. \tag{16.68}$$

For a side face perpendicular to the hemicube's x axis, as shown in Fig. 16.68(b), we have

$$r = \sqrt{y_p^2 + z_p^2 + 1}, \tag{16.69}$$

$$\cos\theta_i = \frac{z_p}{r}, \qquad \cos\theta_p = \frac{1}{r}.$$

Here, Eq. (16.66) simplifies to

$$\Delta F_p = \frac{z_p}{\pi(y_p^2 + z_p^2 + 1)^2}\Delta A. \tag{16.70}$$

Because of symmetry, the values of ΔF_p need to be computed for only one-eighth of the top face and one-quarter of a single side half face.

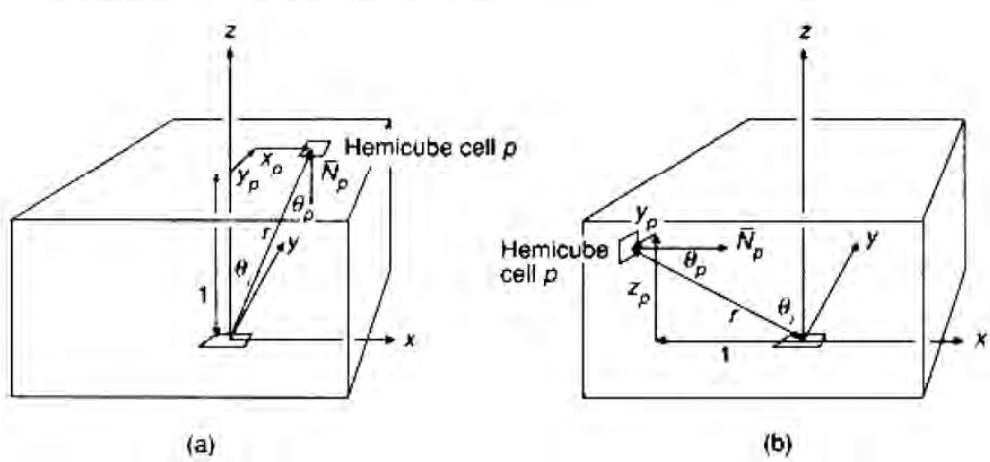


Fig. 16.68 Delta form factors. (a) The top face. (b) A side face. (After [COHE85].)

We can approximate F_{di-j} for any patch j by summing the values of ΔF_p associated with each cell p in A_j's hemicube projections. (Note that the values of ΔF_p for all the hemicube's cells sum to 1.) Assuming that the distance between the patches is large relative to the size of the patches, these values for F_{di-j} may be used as the values of F_{i-j} in Eq. (16.62) to compute the patch radiosities. Color Plate III.21(a–b) was made with the hemicube algorithm. Because much of the computation performed using the hemicube involves computing item buffers, it can take advantage of existing z-buffer hardware. On the other hand, because it uses image-precision operations, the hemicube is prone to aliasing.

Nishita and Nakamae [NISH85a] have adopted a different approach to computing form factors in occluded environments by incorporating their shadow algorithm for area light sources (Section 16.8) into a radiosity algorithm that was used to make Color Plate III.22(a–b). Color Plate III.22(c) [NISH86] adds to this a model of sky light, approximated by a large hemispherical source of diffuse light. The hemisphere is divided into bands that are transversely uniform and longitudinally nonuniform. As with other luminous surfaces, the effects of occluding objects are modeled.

16.13.3 Substructuring

The finer the patch parametrization, the better the results, at the expense of increased computation time for n^2 form factors. To prevent this square-law increase in the number of form factors, Cohen, Greenberg, Immel, and Brock [COHE86] adaptively subdivide patches into subpatches at places in the patch mesh at which a high radiosity gradient is found. The subpatches created by this *substructuring* process are not treated like full-fledged patches. Whenever a patch i is subdivided into subpatches, the form factors F_{s-j} from each subpatch s to each patch j are computed using the hemicube technique, but form factors from the patches to the subpatches are not computed. After a patch has been broken into subpatches, however, the previously calculated values of each form factor from the patch to other patches are replaced by the more accurate area-weighted average of the form factors from its m subpatches:

$$F_{i-j} = \frac{1}{A_i} \sum_{1 \le s \le m} F_{s-j} A_s. \tag{16.71}$$

After patch radiosities are calculated as described before, the radiosity of each subpatch s of patch i can be computed as

$$B_s = E_i + \rho_i \sum_{1 \le j \le n} B_j F_{s-j}. \tag{16.72}$$

The algorithm iterates, adaptively subdividing subpatches at places of high radiosity gradient, until the differences reach an acceptable level. The final subpatch radiosities are then used to determine the vertex radiosities. Color Plate III.21(b), made using a nonadaptive version of the algorithm in which subdivision is specified by the user, shows that the same image takes substantially less time to compute when patches are divided into one level of subpatches, than when an equivalent number of patches are used. The adaptive version of the algorithm is initialized with a "first guess" subdivision specified by the user. Color Plate III.21(c) was created by adaptively subdividing the subpatches of Color Plate III.21(b). Note the improved shadow resolution about the table's legs.

Substructuring allows subpatch radiosities to be determined without changing the size of the matrix to be solved in Eq. (16.62). Note that a subpatch's contribution to other patches is still approximated coarsely by its patch's radiosity, but this is a second-order effect in diffuse environments. In a similar fashion, texture mapping can be implemented by computing a single average reflectivity value for a texture-mapped patch that is used for the radiosity computations [COHE86]. When each pixel in the texture-mapped surface is finally rendered, its shade is scaled by the ratio of the texture-map reflectivity value computed for the pixel and the average reflectivity used for the patch.

16.13.4 Progressive Refinement

Given the high costs of executing the radiosity algorithm described thus far, it makes sense to ask whether it is possible to approximate the algorithm's results incrementally. Can we produce a useful, although perhaps inaccurate, image early on, which can be successively refined to greater accuracy as more time is allocated? The radiosity approach described in the previous sections will not let us do this for two reasons. First, an entire Gauss–Seidel iteration must take place before an estimate of the patch radiosities becomes available. Second, form factors are calculated between all patches at the start and must be stored throughout the computation, requiring $O(n^2)$ time and space. Cohen, Chen, Wallace, and Greenberg [COHE88] have developed a progressive-refinement radiosity algorithm that addresses both of these issues.

Consider the approach described thus far. Evaluating the ith row of Eq. (16.62) provides an estimate of patch i's radiosity, B_i, expressed in Eq. (16.60), based on the estimates of the other patch radiosities. Each term of the summation in Eq. (16.60) represents patch j's effect on the radiosity of patch i:

$$B_i \text{ due to } B_j = \rho_i B_j F_{i-j}, \qquad \text{for all } j. \tag{16.73}$$

Thus, this approach *gathers* the light from the rest of the environment. In contrast, the progressive-refinement approach *shoots* the radiosity from a patch into the environment. A straightforward way to do this is to modify Eq. (16.73) to yield

$$B_j \text{ due to } B_i = \rho_j B_i F_{j-i}, \qquad \text{for all } j. \tag{16.74}$$

Given an estimate of B_i, the contribution of patch i to the rest of the environment can be determined by evaluating Eq. (16.74) for each patch j. Unfortunately, this will require knowing F_{j-i} for each j, each value of which is determined with a separate hemicube. This imposes the same overwhelmingly large space–time overhead as does the original approach. By using the reciprocity relationship of Eq. (16.59), however, we can rewrite Eq. (16.74) as

$$B_j \text{ due to } B_i = \rho_j B_i F_{i-j} \frac{A_i}{A_j}, \qquad \text{for all } j. \tag{16.75}$$

Evaluating this equation for each j requires only the form factors calculated using a single hemicube centered about patch i. If the form factors from patch i can be computed quickly (e.g., by using z-buffer hardware), then they can be discarded as soon as the radiosities shot

from patch i have been computed. Thus, only a single hemicube and its form factors need to be computed and stored at a time.

As soon as a patch's radiosity has been shot, another patch is selected. A patch may be selected to shoot again after new light has been shot to it from other patches. Therefore, it is not patch i's total estimated radiosity that is shot, but rather ΔB_i, the amount of radiosity that patch i has received since the last time that it shot. The algorithm iterates until the desired tolerance is reached. Rather than choose patches in random order, it makes sense to select the patch that will make the most difference. This is the patch that has the most energy left to radiate. Since radiosity is measured per unit area, a patch i is picked for which $\Delta B_i A_i$ is the greatest. Initially, $B_i = \Delta B_i = E_i$ for all patches, which is nonzero only for light sources. The pseudocode for a single iteration is shown in Fig. 16.69.

Each execution of the pseudocode in Fig. 16.69 will cause another patch to shoot its unshot radiosity into the environment. Thus, the only surfaces that are illuminated after the first execution are those that are light sources and those that are illuminated directly by the first patch whose radiosity is shot. If a new picture is rendered at the end of each execution, the first picture will be relatively dark, and those following will get progressively brighter. To make the earlier pictures more useful, we can add an ambient term to the radiosities. With each additional pass through the loop, the ambient term will be decreased, until it disappears.

One way to estimate the ambient term uses a weighted sum of the unshot patch radiosities. First, an average diffuse reflectivity for the environment, ρ_{avg} is computed as a weighted sum of the patch diffuse reflectivities,

$$\rho_{avg} = \sum_{1 \le i \le n} \rho_i A_i \Big/ \sum_{1 \le i \le n} A_i. \tag{16.76}$$

This equation is used to compute an overall reflection factor R, intended to take into account the different reflected paths through which energy can travel from one patch to another,

$$R = 1 + \rho_{avg} + \rho_{avg}^2 + \rho_{avg}^3 + \ldots = \frac{1}{1 - \rho_{avg}}. \tag{16.77}$$

```
select patch i;

calculate F_{i,j} for each patch j;

for (each patch j) {
    ΔRadiosity = ρ_j ΔB_i F_{i,j} (A_i / A_j);
    ΔB_j += ΔRadiosity;
    B_j += ΔRadiosity;
}

ΔB_i = 0;
```

Fig. 16.69 Pseudocode for shooting radiosity from a patch

```
for (each patch i) {
    ΔB_i = E_i;
    for (each subpatch s in i)
        B_s = E_i;
}

AreaSum = Σ_{1≤i≤n} A_i;

Ambient = R Σ_{1≤i≤n} (ΔB_i A_i)/AreaSum;

while (not converged) {
    select patch i with greatest ΔB_i A_i;
    determine F_{i→s} for all subpatches s in all patches;

    /* ΔEnergy is initialized to the total energy shot. */
    ΔEnergy = ΔB_i A_i;

    /* Shoot radiosity from patch i. */
    for (each patch j seen by i) {
        OldΔB = ΔB_j;
        for (each subpatch s in j seen by i) {
            ΔRadiosity = ρ_j ΔB_i F_{i→s} A_s/A_s;
            B_s += ΔRadiosity;
            ΔB_j += ΔRadiosity A_s/A_j;
        }
        /* Decrement ΔEnergy by total energy gained by patch j. */
        ΔEnergy -= (ΔB_j - OldΔB) A_j;
    }

    determine vertex radiosities from subpatch radiosities, using
        B_s + ρ_j Ambient as radiosity of subpatch s of patch j;
    if (radiosity gradient between adjacent vertices is too high)
        subdivide offending subpatches and reshoot from patch i to them;
    ΔB_i = 0;

    perform view-dependent visible-surface determination and shading;

    /* Use ΔEnergy (energy absorbed by patches hit) to determine new value of Ambient. */
    Ambient -= R ΔEnergy / AreaSum;
} /* while */
```

Fig. 16.70 Pseudocode for progressive-refinement radiosity method with ambient light and substructuring.

Each patch's unshot radiosity is weighted by the ratio of the patch's area to the environment's area, providing an approximation to the form factor from an arbitrary differential area to that patch. Thus, the estimate of the ambient term accounting for unshot radiosity is

$$Ambient = R \sum_{1 \le i \le n} (\Delta B_i A_i) / \sum_{1 \le i \le n} A_i. \tag{16.78}$$

This ambient term is used to augment the patch's radiosity for display purposes only, yielding

$$B'_i = B_i + \rho_i Ambient. \tag{16.79}$$

Figure 16.70 shows the pseudocode for the entire algorithm. Substructuring is provided by shooting radiosity from patches to subpatches to determine subpatch radiosities. Thus, hemicubes are created for patches, but not for subpatches. Adaptive subdivision is accomplished by subdividing a patch further when the radiosity gradient between adjacent subpatch vertices is found to be too high. Color Plate III.23, which is rendered using an ambient term, depicts stages in the creation of an image after 1, 2, 24, and 100 iterations.

16.13.5 Computing More Accurate Form Factors

Although the use of fast z-buffer hardware makes the hemicube an efficient algorithm, the technique has a number of failings [BAUM89; WALL89]:

- Recall that the identity of only one patch is stored per hemicube pixel. Therefore, a grid of patches may alias when projected onto a side of the hemicube, just as they would when processed with a z-buffer algorithm. This can show up as a regular pattern of patches that are not represented in the hemicube. Furthermore, a patch that is small when projected on the hemicube may be large when projected on the image plane.
- Use of the hemicube assumes that the center point of a patch is representative of the patch's visibility to other patches. If this assumption is shown to be untrue, the surface can be broken up into subpatches, but there is only a single subdivision granularity for the patch; the same patch cannot be subdivided to different levels for different patches that it views.
- Patches must be far from each other for the hemicube approach to be correct. This is a serious problem if two patches are adjacent, since all calculations are done from the center of the hemicube, the form factor will be underestimated, because the calculations do not take into account the proximity of the adjacent parts of the patches.

A progressive radiosity approach developed by Wallace, Elmquist, and Haines [WALL89] uses ray tracing to evaluate form factors, instead of the hemicube. When a source patch is to shoot its radiosity, rays are fired from each vertex in the scene to the source to compute the form factor from the source to the vertex. This is accomplished by decomposing the source patch into a number of small finite subareas, each of which is the target of a ray shot from a vertex. If the ray is not occluded, then the target is visible, and the form factor between the differential vertex and the finite area target is computed, using an analytic expression, based on some simplifying geometric assumptions. If desired, the

ray intersection calculations can be performed with a resolution-independent true curved-surface database, so that the time for an individual ray test does not depend on the number of polygons. The form factor between the vertex and the entire source is computed as an area-weighted average of the form factors between each subarea and the vertex, and the result is used to compute the contribution of the source to the vertex. This approach has a number of advantages. Radiosities are computed at the vertices themselves, where they are ultimately needed for shading. Vertex normals can be used, allowing polygonal meshes that approximate curved surfaces. Nonphysical point light sources can be handled by tracing a single ray to the light source and using its illumination equation to determine the irradiance at each vertex. The number of areas into which a source is decomposed and whether rays are actually fired (i.e., if shadow testing is to be performed) can all be determined individually for each vertex. Color Plates III.24 and III.25 were created using this algorithm.

A contrasting approach to solving inaccuracies caused by the hemicube is taken by Baum, Rushmeier, and Winget [BAUM89]. They recognize that the hemicube form factors often are accurate; therefore, they have developed error-analysis tests to choose, for each patch, when to use the hemicube, when to subdivide the patch further, and when to use a more expensive, but more accurate, analytic technique for computing form factors.

16.13.6 Specular Reflection

The radiosity methods described so far treat only diffuse reflection. Therefore, all of a patch's radiosity may be treated uniformly when it is dispersed to other patches: The radiosity leaving a patch in any direction is influenced by the patch's total radiosity, not by the directions from which its incoming energy was acquired. Immel, Cohen, and Greenberg [IMME86] extended the radiosity method to model specular reflection. Rather than compute a single radiosity value for each patch, they partition the hemisphere over the patch into a finite set of solid angles, each of which establishes a direction for incoming or outgoing energy. Given the patch's bidirectional reflectivity (Section 16.7), they compute the outgoing radiosity in each direction in terms of its emittance in that direction and the incident light from each of the set of directions, weighting each direction's contribution accordingly. Finally, they render a picture from intensities that are determined at each vertex by using the direction from the vertex to the eye to interpolate among the closest directional radiosities. Although the results shown in Color Plate III.26 are promising, the approach has a tremendous overhead in both time and space, which will only increase if highly specular surfaces are modeled. One solution is to combine a radiosity method with ray tracing.

16.13.7 Combining Radiosity and Ray Tracing

Consider the tradeoffs between radiosity methods and ray tracing. Radiosity methods are well suited to diffuse reflection because a diffuse surface's bidirectional reflectivity is constant in all outgoing directions. Thus, all radiosities computed are view-independent. On the other hand, the pure radiosity method for specular surfaces described previously is not practical, because specular reflection from a surface is highly dependent on the angle with which an observer (or another surface) views the surface. Therefore, much extra

information must be computed, because no information about the desired view is provided. In addition, this directional information is discretized and must be interpolated to accommodate a specific view. Not only does the interpolation make it impossible to model sharp reflections, but also the sampling performed by the discretization can result in aliasing.

In contrast, ray tracing calculates specular reflections well, since the eyepoint is known in advance. Although conventional ray tracing does not model global diffuse phenomena, some of the approaches discussed in Section 16.12.4 do. Correctly solving for the diffuse reflection from a piece of surface requires that all the surfaces with which a surface exchanges energy be taken into account; in short, it requires a radiosity method.

It makes sense to combine ray tracing and radiosity to take advantage of ray tracing's ability to model specular phenomena and of the radiosity method's ability to model diffuse interactions. Unfortunately, simply summing the pixel values computed by a diffuse radiosity method and a specular ray tracer will not suffice. For example, the diffuse radiosity method will fail to take into account the extra illumination falling on a diffuse surface from a specular surface. It is necessary to account for transfer from diffuse to diffuse, diffuse to specular, specular to diffuse, and specular to specular reflection.

Wallace, Cohen, and Greenberg [WALL87] describe a two-pass approach that combines a view-independent radiosity method, executed in the first pass, with a view-dependent ray-tracing approach, executed in the second pass. As mentioned previously, the first pass must take into account specular, as well as diffuse, reflection. If only perfect, mirrorlike specular reflection is allowed, this can be supported by reflecting each patch about the plane of a specular surface [RUSH86]. Each specular patch is thus treated as a window onto a "mirror world." The form factor from a patch to one of these mirror reflections accounts for the specular reflection from the patch that is doing the mirroring.

In the second view-dependent pass, a *reflection frustum* is erected at each point on a surface that corresponds to a pixel in the image. As shown in Fig. 16.71, the reflection frustum consists of a little z-buffer, positioned perpendicular to the reflection direction, and covering the small incoming solid angle that is most significant for the surface's

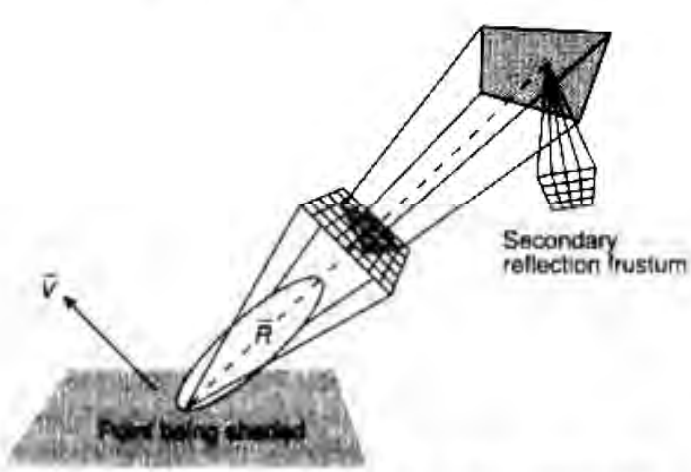


Fig. 16.71 The reflection frustum. (After [WALL87].)

bidirectional reflectivity. The patches are z-buffered onto the frustum, using Gouraud shading to interpolate the patches' first-pass diffuse intensities across their projections. A ray is traced recursively through each pixel on the frustum that sees a specular surface, spawning a new reflection frustum at each intersection. The values computed for each frustum pixel are then weighted to model the surface's ρ_s. The second pass thus uses specular transport to combine radiosities determined during the first pass. Transparency can be accommodated by erecting a transmission frustrum in the direction of refraction. The image on the cover of this book (Color Plate I.9) was created using this algorithm.

The mirror-world approach used in the first pass handles only perfect specular reflection and results in a proliferation of form factors. Shao, Peng, and Liang [SHAO88] have implemented a two-pass approach that allows Phong-like bidirectional reflectance functions in the first pass, without the need to duplicate patches.

Sillion and Puech [SILL89] extend the two-pass technique to calculate *extended form factors* in the first pass that model any number of specular reflections or refractions. Rather than proliferating mirror-reflection form factors, they instead use recursive ray tracing to compute the form factors, as well as in the view-dependent second pass. Color Plate III.27 demonstrates why the diffuse first pass must take specular reflection into account. Color Plate III.27(a) shows the results of a conventional diffuse radiosity approach. (The part of the table near the mirror is lit by light diffusely reflected from the inside of the tall lamp.) The conventional diffuse first pass was augmented with a pure ray-tracing second pass to produce Color Plate III.27(b), which includes a specular reflection from the mirror. In contrast, Color Plate III.27(c) shows the results of Sillion and Puech's two-pass approach. It shares the same ray-tracing second pass as Color Plate III.27(b), but uses extended form factors in the first pass. Each surface acts as a diffuse illuminator in the first pass, but the use of the extended form factors means that the diffusely emitted energy takes specular interreflection into account. Note the light specularly reflected from the mirror onto the table and the back of the vase during the first pass. Color Plate III.28 is a more complex example that includes a reflecting sphere.

16.14 THE RENDERING PIPELINE

Now that we have seen a variety of different ways to perform visible-surface determination, illumination, and shading, we shall review how these processes fit into the standard graphics pipeline introduced in Chapter 7 and depicted in Fig. 7.26. For simplicity, we assume polygonal environments, unless otherwise specified. Chapter 18 provides a more detailed discussion of how some of these pipelines may be implemented in hardware.

16.14.1 Local Illumination Pipelines

z-buffer and Gouraud shading. Perhaps the most straightforward modification to the pipeline occurs in a system that uses the z-buffer visible-surface algorithm to render Gouraud-shaded polygons, as shown in Fig. 16.72. The z-buffer algorithm has the advantage that primitives may be presented to it in any order. Therefore, as before, primitives are obtained by traversing the database, and are transformed by the modeling transformation into the WC system.

Primitives may have associated surface normals that were specified when the model was built. Since the lighting step will require the use of surface normals, it is important to

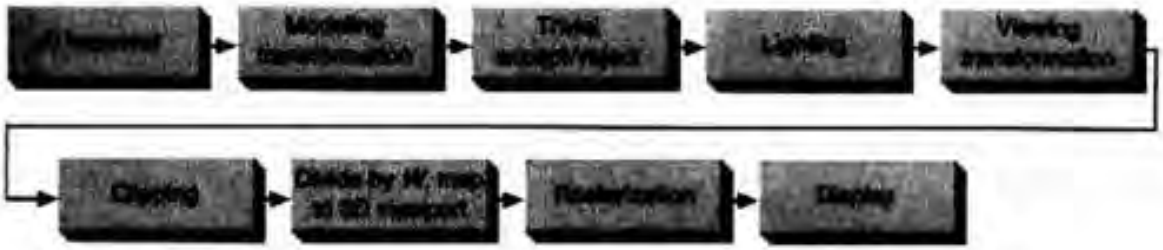


Fig. 16.72 Rendering pipeline for z-buffer and Gouraud shading.

remember that normals must be transformed correctly, using the methods discussed in the Appendix. Furthermore, we cannot just ignore stored normals and attempt to recompute new ones later using the correctly transformed vertices. The normals defined with the objects may represent the true surface geometry, or may specify user-defined surface blending effects, rather than just being the averages of the normals of shared faces in the polygonal mesh approximation.

Our next step is to cull primitives that fall entirely outside of the window and to perform back-face culling. This trivial-reject phase is typically performed now because we want to eliminate unneeded processing in the lighting step that follows. Now, because we are using Gouraud shading, the illumination equation is evaluated at each vertex. This operation must be performed in the WC system (or in any coordinate system isometric to it), before the viewing transformation (which may include skew and perspective transformations), to preserve the correct angle and distance from each light to the surface. If vertex normals were not provided with the object, they may be computed immediately before lighting the vertices. Culling and lighting are often performed in a lighting coordinate system that is a rigid body transformation of WC (e.g., VRC when the view orientation matrix is created with the standard PHIGS utilities).

Next objects are transformed to NPC by the viewing transformation, and clipped to the view volume. Division by W is performed, and objects are mapped to the viewport. If an object is partially clipped, correct intensity values must be calculated for vertices created during clipping. At this point, the clipped primitive is submitted to the z-buffer algorithm, which performs rasterization, interleaving scan conversion with the interpolation needed to compute the z value and color-intensity values for each pixel. If a pixel is determined to be visible, its color-intensity values may be further modified by depth cueing (Eq. 16.11), not shown here.

Although this pipeline may seem straightforward, there are many new issues that must be dealt with to provide an efficient and correct implementation. For example, consider the problems raised by handling curved surfaces, such as B-spline patches, which must be tessellated. Tessellation should occur after transformation into a coordinate system in which screen size can be determined. This enables tessellation size to be determined adaptively, and limits the amount of data that is transformed. On the other hand, tessellated primitives must be lit in a coordinate system isometric to world coordinates. Abi-Ezzi [ABIE89] addresses these issues, proposing a more efficient, yet more complex, formulation of the pipeline that incorporates feedback loops. This new pipeline uses a lighting coordinate system that is an isometric (i.e., rigid or Euclidean) transformation of WC, yet is computationally close to DC to allow tessellation decisions to be made efficiently.

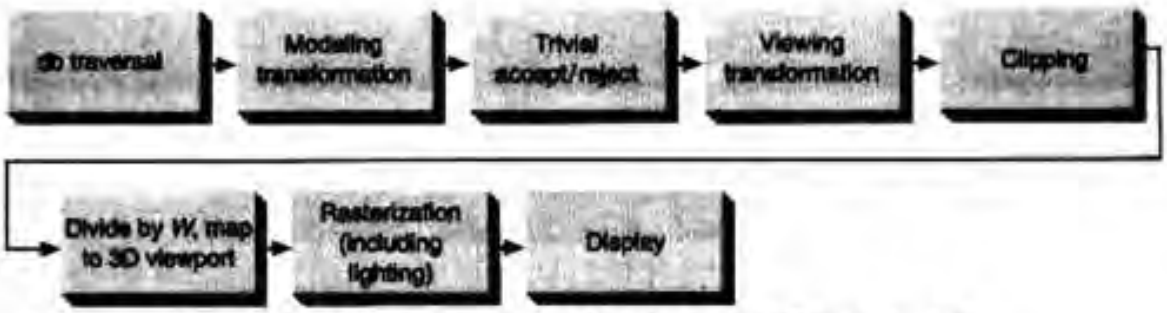


Fig. 16.73 Rendering pipeline for z-buffer and Phong shading.

z-buffer and Phong shading. This simple pipeline must be modified if we wish to accommodate Phong shading, as shown in Fig. 16.73. Because Phong shading interpolates surface normals, rather than intensities, the vertices cannot be lit early in the pipeline. Instead, each object must be clipped (with properly interpolated normals created for each newly created vertex), transformed by the viewing transformation, and passed to the z-buffer algorithm. Finally, lighting is performed with the interpolated surface normals that are derived during scan conversion. Thus, each point and its normal must be backmapped into a coordinate system that is isometric to WC to evaluate the illumination equation.

List-priority algorithm and Phong shading. When a list-priority algorithm is used, primitives obtained from traversal and processed by the modeling transformation are inserted in a separate database, such as a BSP tree, as part of preliminary visible-surface determination. Figure 16.74 presents the pipeline for the BSP tree algorithm, whose preliminary visible-surface determination is view-independent. As we noted in Chapter 7, the application program and the graphics package may each keep separate databases. Here, we see that rendering can require yet another database. Since, in this case, polygons are split, correct shading information must be determined for the newly created vertices. The rendering database can now be traversed to return primitives in a correct, back-to-front order. The overhead of building this database can, of course, be applied toward the creation of multiple pictures. Therefore, we have shown it as a separate pipeline whose output is a new database. Primitives extracted from the rendering database are clipped and normalized, and are presented to the remaining stages of the pipeline. These stages are structured much like those used for the z-buffer pipeline, except that the only visible-surface process they need to perform is to guarantee that each polygon will correctly overwrite any

Fig. 16.74 Rendering pipeline for list-priority algorithm and Phong shading.

Plate III.1 Grass and trees. 2×10^7 objects, organized in a hierarchy of lists and grids (Section 15.10.2). Ray traced at 512×512 resolution with 16 rays/pixel, taking 16 hours on an IBM 4381/Group 12. (John Snyder and Alan Barr, © Caltech Computer Graphics Group.)

Plate III.2 Ray tracing CSG (Section 15.10.3). Bowl with stars and moons. (a) Objects used in CSG operations. (b) Cross-section of (a). (c) Bowl formed by CSG operations. Red sphere is truncated by intersection with blue cylinder, and result is hollowed by subtracting internal grey sphere visible in (b). Extruded moons (formed by subtracting green cylinders from orange cylinders) and extruded white stars cut holes in bowl when subtracted from earlier result. Reflection mapping (Section 16.6) and Cook–Torrance illumination model (Section 16.7) are used to give bowl a metallic appearance. (Courtesy of David Kurlander, Columbia University.)

(a)

(b)

(c)

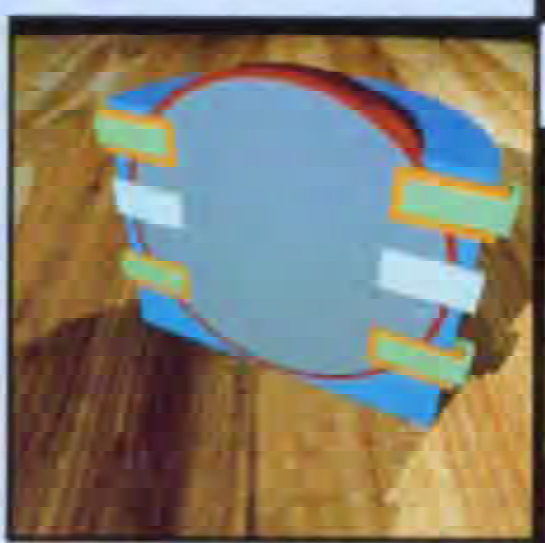


Plate III.3 A torus bump mapped with a hand-generated bump function (Section 16.3.3). (By Jim Blinn. Courtesy of University of Utah.)

Plate III.4 A strawberry bump mapped with a hand-generated bump function (Section 16.3.3). (By Jim Blinn. Courtesy of University of Utah.)

Plate III.5 Objects with shadows generated by two-pass object-precision algorithm of Section 16.4.2. (a) One light source. (b) Two light sources. (Peter Atherton, Kevin Weiler, and Donald P. Greenberg, Program of Computer Graphics, Cornell University, 1978.)

(a) (b)

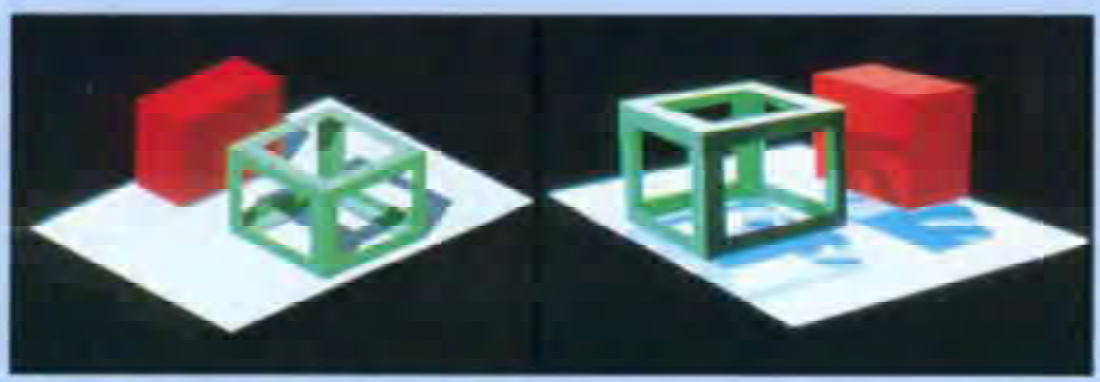


Plate III.6 Room with shadows generated by object-precision shadow-volume BSP tree algorithm of Section 16.4.3. (a) Scene with two point light sources. (b) Same scene with black lines indicating polygon fragmentation. Dark gray fragments are lit by no lights, light gray fragments are lit by one light, and non-gray fragments are lit by both lights. (Courtesy of Norman Chin, Columbia University.)

(a)

(b)

Plate III.7 Nonrefractive transparency using extended z-buffer algorithm of Section 16.5.1. Unterlafette database is courtesy of CAM-I (Computer Aided Manufacturing International, Inc., Arlington, TX). (Rendered on a Stardent 1000 by Abraham Mammen.)

(a) (b)

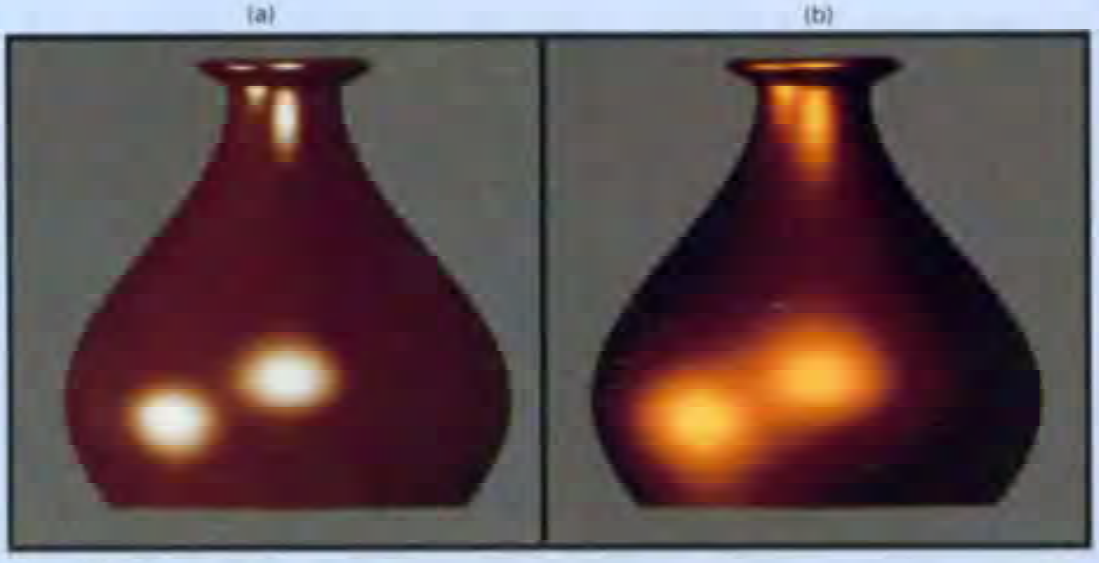
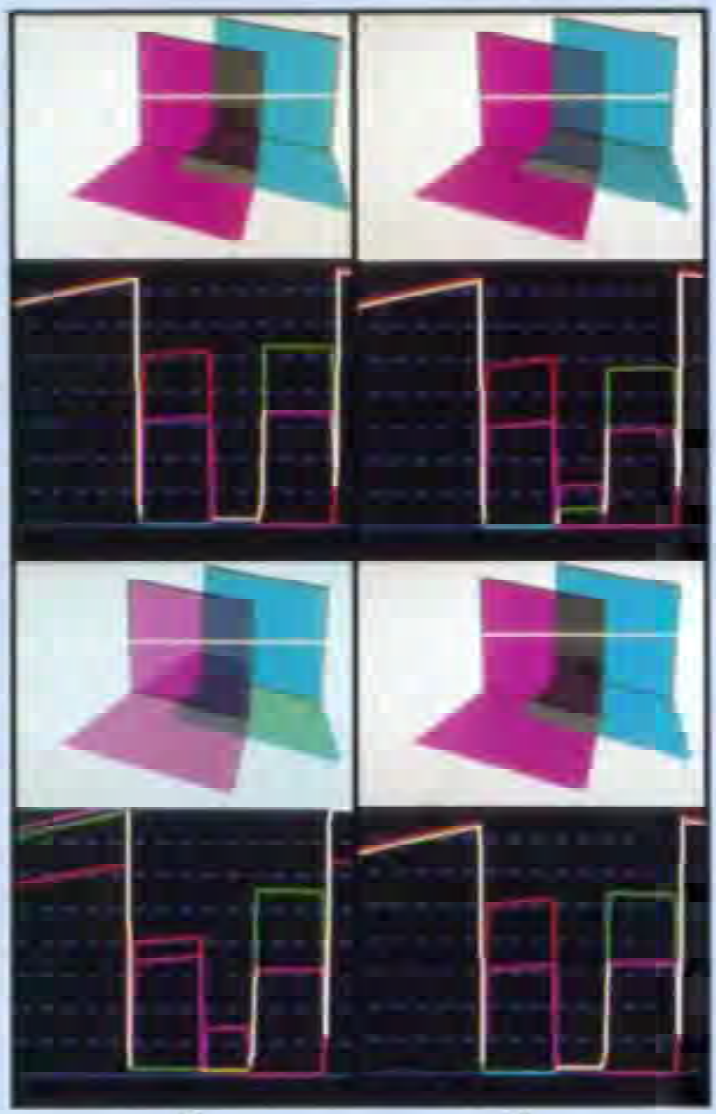


Plate III.8 Two vases rendered with the Cook-Torrance illumination model (Section 16.7). Both are lit by two lights with $I_{l_1} = I_{l_2} =$ CIE standard illuminant D6500, $d\omega_{l_1} = 0.0001$, and $d\omega_{l_2} = 0.0002$; $I_a = 0.01 I_{l_1}$; ρ_d = the bidirectional reflectivity of copper for normal incidence; $\rho_s = \pi\rho_d$. (a) Copper-colored plastic: $k_s = 0.1$; F = reflectivity of a vinyl mirror; D = Beckmann function with $m = 0.15$; $k_d = 0.9$. (b) Copper metal: $k_s = 1.0$; F = reflectivity of a copper mirror; D = Beckmann function with $m_1 = 0.4$, $w_1 = 0.4$, $m_2 = 0.2$, $w_2 = 0.6$; $k_d = 0.0$. (By Robert Cook, Program of Computer Graphics, Cornell University.)

(a) (b)



(c) (d)

Plate III.9 Comparison of spectral sampling techniques for two overlapping filters (Section 16.9). Plots show RGB values for marked scan line with red (R), green (G), and magenta (B) lines. (a) One sample per nm from 360–830 nm. (b) 3 CIE XYZ samples. (c) 3 RGB samples. (d) 9 spectral samples. (Roy A. Hall and Donald P. Greenberg, Program of Computer Graphics, Cornell University, 1983.)

Plate III.10 Spheres and checkerboard. An early image produced with recursive ray tracing (Section 16.12). (Courtesy of Turner Whitted, Bell Laboratories.)

Plate III.11 Ray-traced images. (a) Scene from short film *Quest* (1985). (Michael Sciulli, James Arvo, and Melissa White. © Hewlett-Packard.) (b) "Haute Air." Functions were used to modify color, surface normals, and transparency at nearly every pixel. (Courtesy of David Kurlander, Columbia University, 1986.)

(a)

(b)

(a)

Plate III.12 Images ray-traced with the light-buffer algorithm (Section 16.12.1). Rendered with $|L| = 50$ light buffers on a VAX 11/780. (a) Glass leaved tree. 768 objects (6 polygons, 256 spheres, 511 quadrics) and 3 lights. A 512 × 480 resolution version was rendered in 412 minutes with light buffers, 1311 minutes without. (b) Kitchen. 224 objects (1298 polygons, 4 spheres, 76 cylinders, 36 quadrics) and 5 lights. A 436 × 479 resolution version was rendered in 246 minutes with light buffers, 602 minutes without. (Eric Haines and Donald P. Greenberg, Program of Computer Graphics, Cornell University, 1986.)

(b)

(a)

(b)

Plate III.13 Comparison of illumination models for ray tracing (Section 16.12.2). Note differences in reflectivity of the base of the dish, and the color of the transparent and reflective spheres. (a) Whitted illumination model. (b) Hall illumination model. (Roy A. Hall and Donald P. Greenberg, Program of Computer Graphics, Cornell University, 1983.)

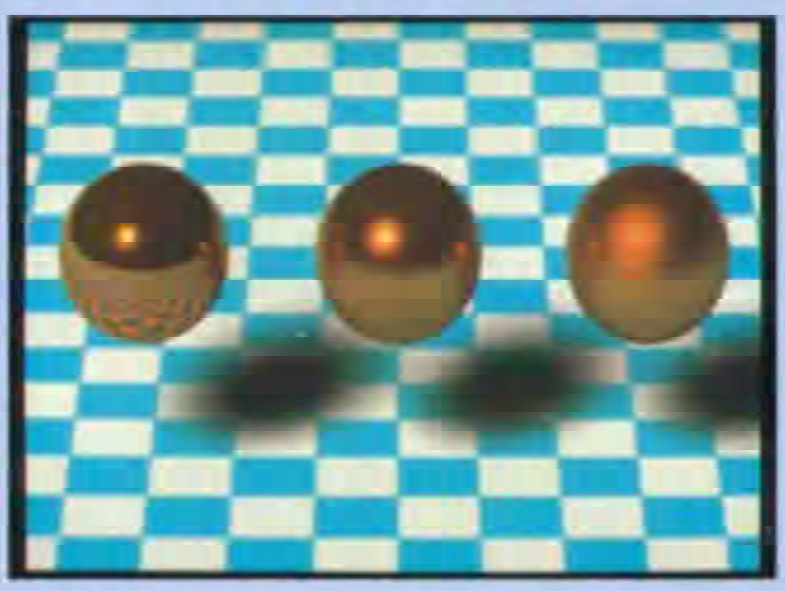


Plate III.14 Cone tracing (Section 16.12.3). Three spheres. Dull reflections are created by increasing the angular spread of reflected rays by 0.0, 0.2, and 0.4 radians, respectively, for spheres from left to right. (Courtesy of John Amanatides, University of Toronto.)

Plate III.15 Beam tracing (Section 16.12.3). A mirrored cube in a texture-mapped room. (Paul Heckbert and Pat Hanrahan, © NYIT 1984.)

Plate III.16 *1984*. Rendered using distributed ray tracing (Section 16.12.4) at 4096 × 3550 pixels with 16 samples per pixel. Note the motion-blurred reflections and shadows with penumbrae cast by extended light sources. (By Thomas Porter. © Pixar 1984. All Rights Reserved.)

Plate III.17 Scene rendered with integral equation method (Section 16.12.4). All opaque objects are Lambertian. Note interobject reflections. Computed at 512 × 512 resolution with 40 paths/pixel in 1221 minutes on an IBM 3081. (J. Kajiya, California Institute of Technology.)

(a)

(b)

Plate III.18 Daylit office rendered by ray tracing with diffuse interreflection (Section 16.12.4). (a) Direct illumination only. (b) 1 diffuse bounce. (c) 7 diffuse bounces. (Courtesy of Greg Ward, Lawrence Berkeley Laboratory.)

(c)

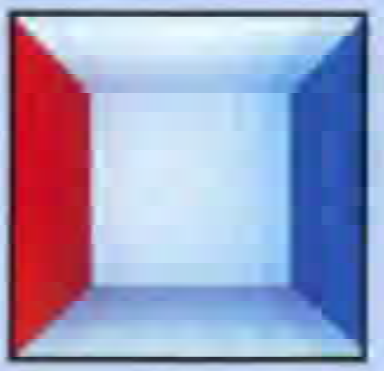
Plate III.19 Conference room. (a) Photograph of actual room. (b) Model rendered by ray tracing, using same software used for Color Plate III.18, but without interreflection calculation. (Courtesy of Greg Ward, Anat Grynberg, and Francis Rubinstein, Lawrence Berkeley Laboratory.)

(a)

(b)

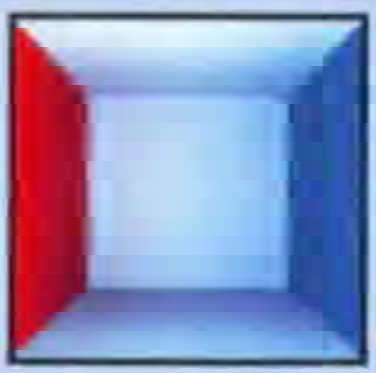
Plate III.20 Radiosity (Section 16.13.1). Cube with six diffuse walls (emissive white front wall is not shown). (a) Photograph of actual cube. (b) Model rendered with 49 patches per side, using constant shading. (c) Model rendered with 49 patches per side, using interpolated shading. (Cindy M. Goral, Kenneth E. Torrance, Donald P. Greenberg, and Bennett Battaile, Program of Computer Graphics, Cornell University, 1984.)

(b)



(a)

(c)



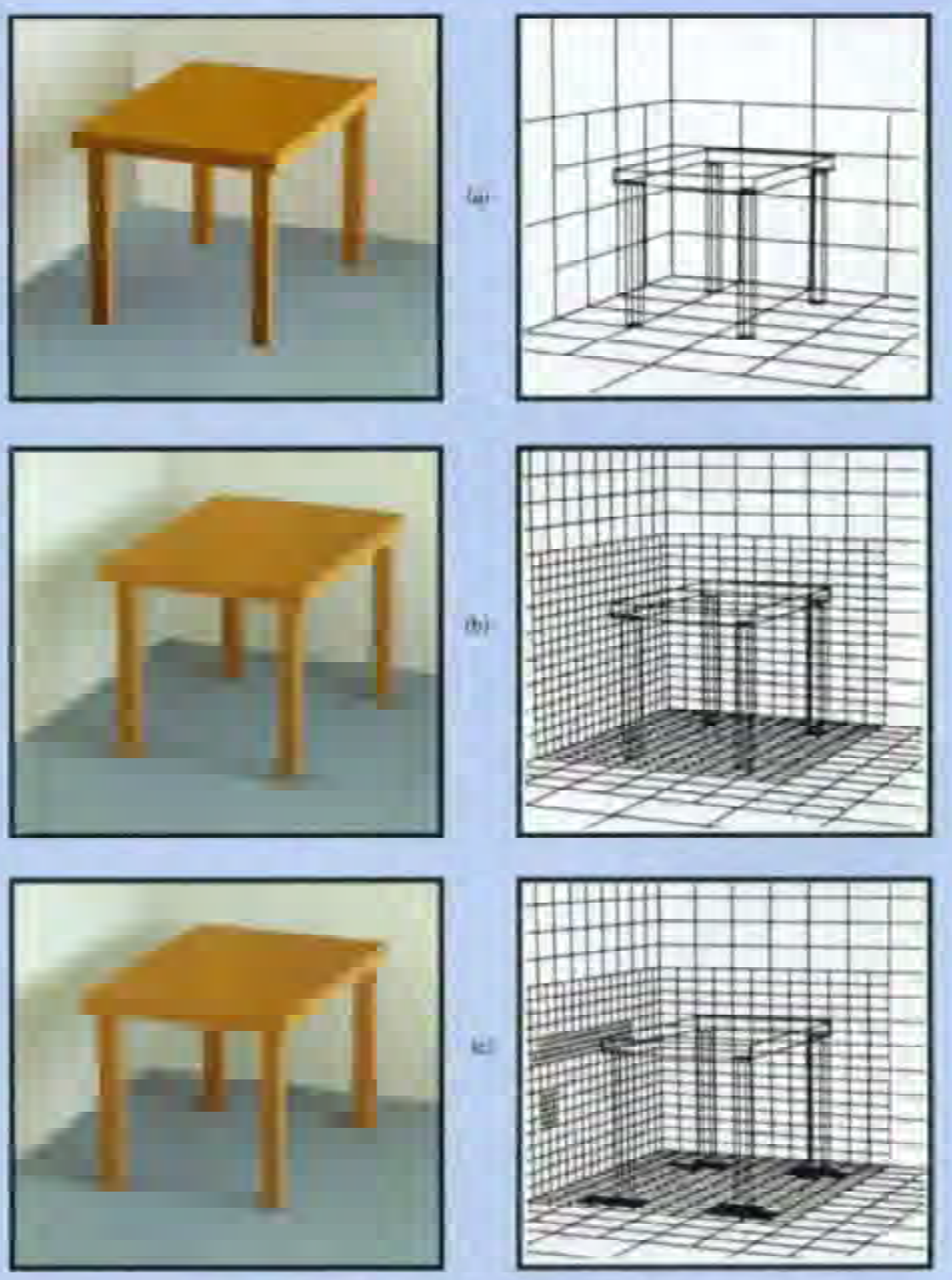


Plate III.21 Radiosity algorithm, using hemicube to compute form factors (Sections 16.13.2–3). (a) Coarse patch solution (145 patches, 10 minutes). (b) Improved solution using either more patches (1021 patches, 175 minutes) or substructuring (1021 subpatches, 62 minutes). (c) Refinement of substructured version of (b) using adaptive subdivision (1306 subpatches, 24 additional minutes). (Michael F. Cohen, Program of Computer Graphics, Cornell University, 1985.)

Plate III.22 Machine room rendered with different lighting models (Sections 16.13.2 and 20.8.2) and shadow volumes for extended lights (Section 16.8). (a) Lit by area light source window. (b) Lit by overhead panel lights. (c) Lit by model of clear skylight. (d) Lit by model including atmospheric scattering. Parts (a–c) use a radiosity algorithm. (T. Nishita (Fukuyama University) and E. Nakamae (Hiroshima University).)

(a)

(b)

(c)

(d)

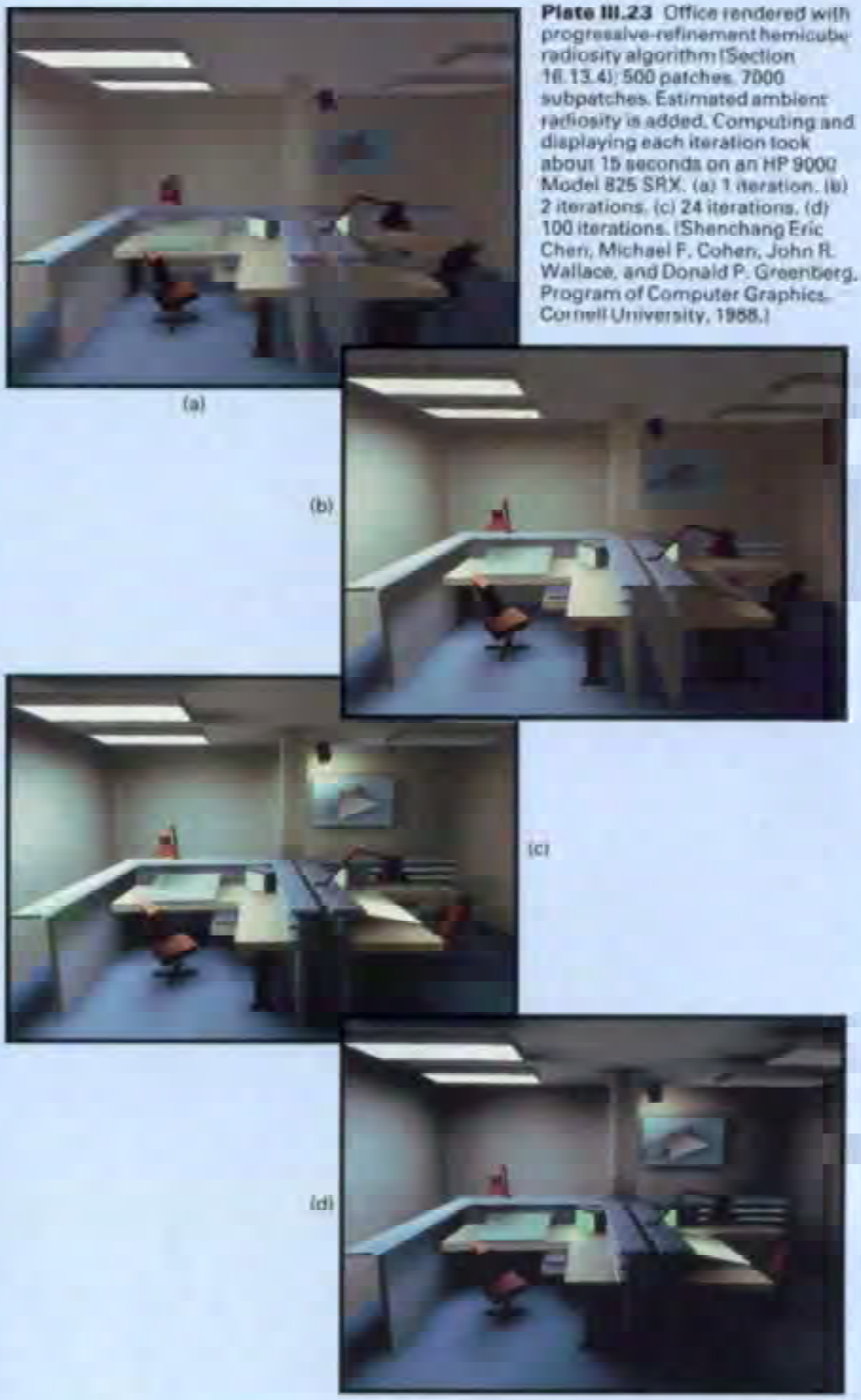


Plate III.23 Office rendered with progressive-refinement hemicube radiosity algorithm (Section 16.13.4); 500 patches, 7000 subpatches. Estimated ambient radiosity is added. Computing and displaying each iteration took about 15 seconds on an HP 9000 Model 825 SRX. (a) 1 iteration, (b) 2 iterations, (c) 24 iterations, (d) 100 iterations. (Shenchang Eric Chen, Michael F. Cohen, John R. Wallace, and Donald P. Greenberg, Program of Computer Graphics, Cornell University, 1988.)

Plate III.24 Nave of Chartres cathedral rendered with progressive-refinement radiosity algorithm, using ray tracing to compute form factors (Section 16.13.6). Two bays, containing 9916 polygons, were processed and copied three more times. Sixty iterations took 59 minutes on HP 9000 Model 835 TurboSRX. (By John Wallace and John Lin, using Hewlett-Packard's Starbase Radiosity and Ray Tracing software. © 1989, Hewlett-Packard Co.)

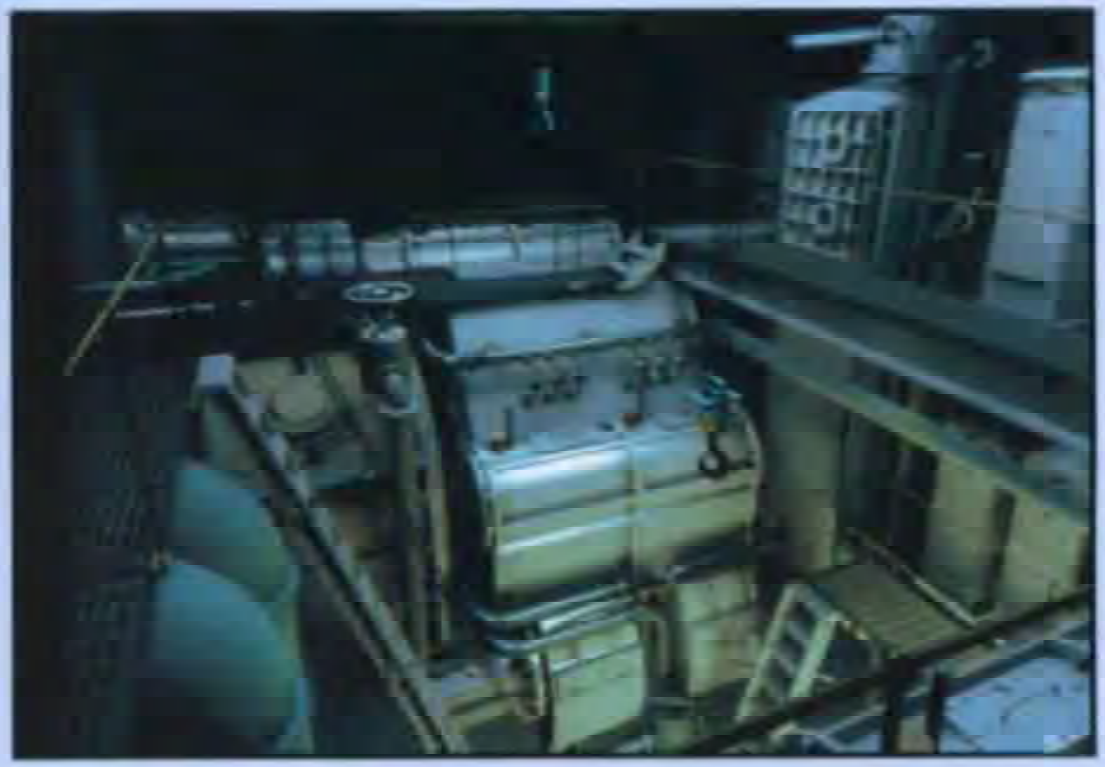


Plate III.25 Boiler room rendered with progressive-refinement radiosity algorithm, using ray tracing to compute form factors. (By John Wallace, John Lin, and Eric Haines, using Hewlett-Packard's Starbase Radiosity and Ray Tracing software. © 1989, Hewlett-Packard Company.)

Plate III.26 Specular radiosity algorithm (Section 16.13.6); 64 specular patches and 237 diffuse patches. (David S. Immel, Michael F. Cohen, Donald P. Greenberg, Program of Computer Graphics, Cornell University 1986.)

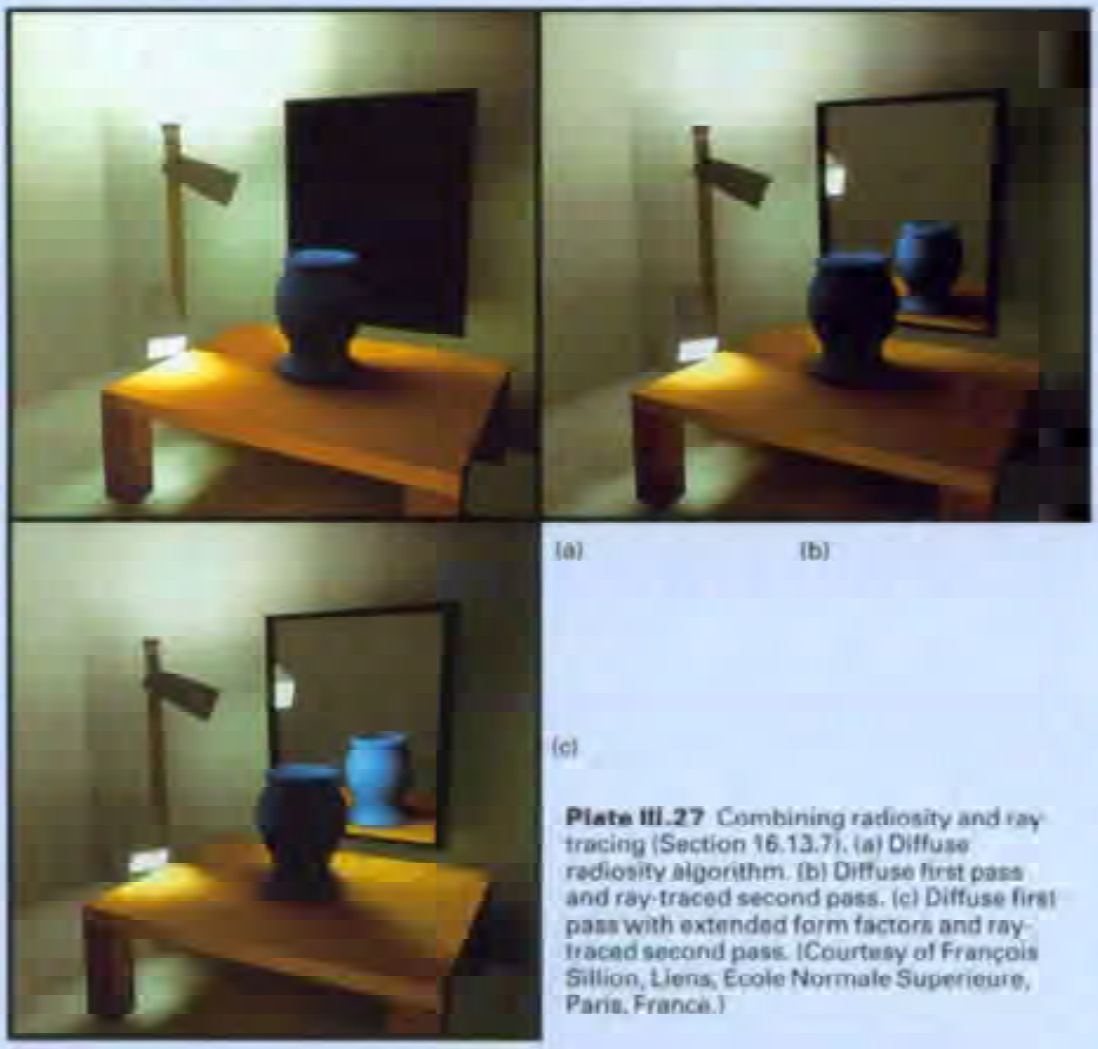


Plate III.27 Combining radiosity and ray tracing (Section 16.13.7). (a) Diffuse radiosity algorithm. (b) Diffuse first pass and ray-traced second pass. (c) Diffuse first pass with extended form factors and ray-traced second pass. (Courtesy of François Sillion, Liens, Ecole Normale Superieure, Paris, France.)

Plate III.28 Room rendered with combined radiosity and ray tracing. (Courtesy of François Sillion, Liens, Ecole Normale Superieure, Paris, France.)

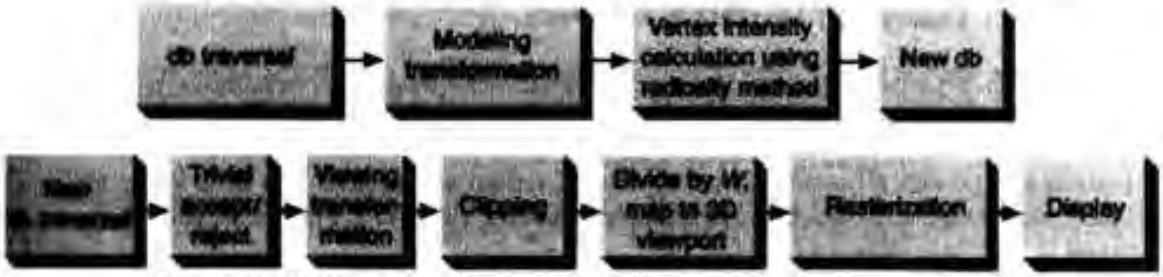


Fig. 16.75 Rendering pipeline for radiosity and Gouraud shading.

previously scan-converted polygon that it intersects. Even this simple overwrite capability is not needed if we instead use an object-precision algorithm that generates a list of fully visible primitives, such as the Weiler–Atherton algorithm.

16.14.2 Global Illumination Pipelines

Thus far, we have ignored global illumination. As we have noted before, incorporating global illumination effects requires information about the geometric relationships between the object being rendered and the other objects in the world. One approach, of which we have seen many examples, is to calculate needed information from a specific viewpoint in advance of scan conversion and to store it in tables (e.g., reflection maps and shadow maps). This eliminates the need to access the full db representation of other objects while processing the current object. In the case of shadows, which depend only on the position of the light source, and not on that of the viewer, preprocessing the environment to add surface-detail polygon shadows is another way to allow the use of an otherwise conventional pipeline.

Radiosity. The diffuse radiosity algorithms offer an interesting example of how to take advantage of the conventional pipeline to achieve global-illumination effects. These algorithms process objects and assign to them a set of view-independent vertex intensities. These objects may then be presented to a modified version of the pipeline for z-buffer and Gouraud shading, depicted in Fig. 16.75, that eliminates the lighting stage.

Ray tracing. Finally, we consider ray tracing, whose pipeline, shown in Fig. 16.76, is the simplest because those objects that are visible at each pixel and their illumination are determined entirely in WC. Once objects have been obtained from the database and transformed by the modeling transformation, they are loaded into the ray tracer's WC database, which is typically implemented using the techniques of Sections 15.10.2 and 16.12.1, to support efficient ray intersection calculations.

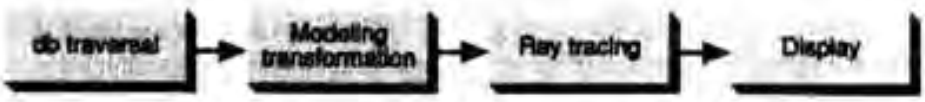


Fig. 16.76 Rendering pipeline for ray tracing.

16.14.3 Designing Flexible Renderers

As we have seen, a wide variety of illumination and shading models has been created. The choice of which to use may be based on concerns as diverse as increasing efficiency, increasing realism, or obtaining visually interesting effects. Simply put, there is no one model that pleases all users. Therefore, several design approaches have been suggested to increase the ease with which illumination and shading algorithms may be implemented and used.

Modularization. A straightforward approach is to modularize the illumination and shading model in a part of the rendering system that is often known as its *shader*. Whitted and Weimer [WHIT82] showed that, by establishing a standard mechanism for passing parameters to shaders, different shaders can be used in the same system: the decision about which shader to call can even be made at run time based on some attribute of the object. Their system performs scan conversion using a scan-line algorithm, and accumulates results as a linked list of spans for each line. Each span contains information about a set of values associated with its endpoints, including their x and z values, and additional information such as normal components, and intensities. The shader interpolates specified values across each span. (Since it typically uses the interpolated z values to perform visible-surface determination with a scan-line z-buffer, the term *shader* is being used quite loosely.)

The Doré graphics system [ARDE89] is designed to offer the programmer additional flexibility. It provides a standard way of expressing a scene database in terms of a set of objects that have methods for performing operations such as rendering, picking, or computing a bounding volume. The display list and its traversal functions form a common core that is intended to make it easy to interface to different rendering systems. A programmer can use the standard set of objects, methods, and attributes, or can design her own using the framework.

Special languages. In contrast to providing extensibility at the level of the programming language in which the system is built, it is possible to design special languages that are better suited to specific graphics tasks. Cook [COOK84a] has designed a special-purpose language in which a shader is built as a tree expression called a *shade tree*. A shade tree is a tree of nodes, each of which takes parameters from its children and produces parameters for its parent. The parameters are the terms of the illumination equation, such as the specular coefficient, or the surface normal. Some nodes, such as *diffuse*, *specular*, or *square root*, are built into the language with which shade trees are specified. Others can be defined by the user and dynamically loaded when needed. All nodes can access information about the lights. Figure 16.77 shows a shade tree for a description of copper. A shade tree thus describes a particular shading process and is associated with one or more objects through use of a separate modeling language. Different objects may have different shade trees, so that an image can be rendered in which a multiplicity of different special-purpose models are mixed. Similarly, in Cook's system, lights and their parameters are defined by light trees, and atmospheric effects, such as haze, are defined by atmosphere trees.

Perlin [PERL85] has developed the notion of a *pixel-stream editor* that takes as input and produces as output arrays of pixels. A pixel is not rigidly defined and may include

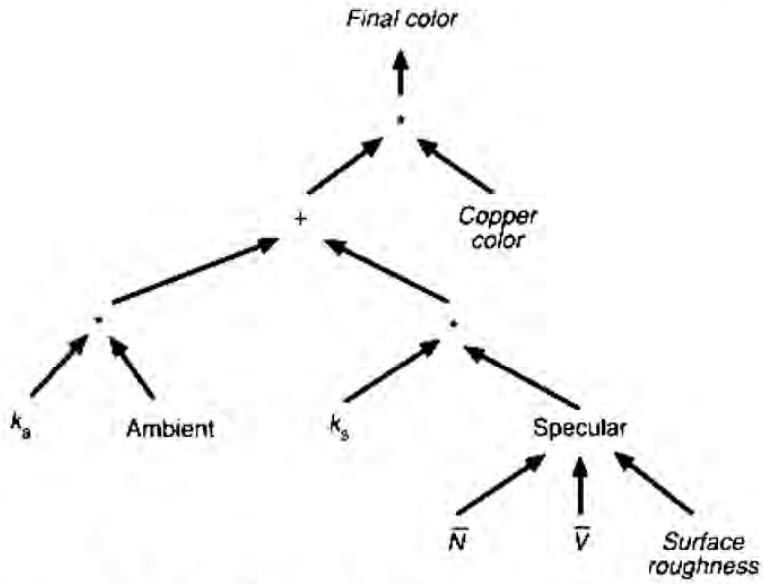


Fig. 16.77 Shade tree for copper. (After [COOK84].)

arbitrary data for a point in the image, such as the material identifier or normal vector at that point. An output pixel need not have the same structure as an input pixel. The pixel-stream editor executes a program written by the user in a high-level language oriented toward pixel manipulation. Thus, the user is encouraged to think of creating images in a series of passes, with intermediate results represented by arrays of pixels that may differ in the kind of information they encode.

The flexibility of shade trees and pixel-stream editors may be combined by designing a rendering system that allows its users to write their own shaders in a special programming language and to associate them with selected objects. This approach is taken in the RenderMan Interface [PIXA88; UPST89], a scene description specification that provides such a shading language. RenderMan defines a set of key places in the rendering process at which user-defined or system-defined shaders can be called. For example, the most common kind of shader, called a *surface shader*, returns the light reflected in a specified direction given a point on the surface, its orientation, and a set of light sources. A user-provided surface shader could implement an illumination equation totally different from those discussed so far. Other shaders include atmosphere shaders that modify the color of light passing between two points, and—in another example of how the word *shader* can be stretched—projection shaders that allow user-defined projections implementing other than parallel or linear perspective projections.

An example. Cook, Carpenter, and Catmull's Reyes Rendering Architecture [COOK87], which was used to produce Color Plates II.24–37, D, and F, provides an interesting example of how to structure a renderer. Reyes chops all objects up into *micropolygons*: small, constant-shaded quadrilaterals that are approximately ½ pixel on a side. This approach, known as *dicing*, occurs along boundaries that are natural for the

object. A patch, for example, is diced parallel to its (s,t) coordinate system. Dicing is performed prior to perspective transformation, based on an estimate of the size of the resulting micropolygons after projection. Much like Catmull's patch subdivision algorithm of Section 15.9, which subdivides patches until they are pixel-sized, Reyes subdivides objects until they are sufficiently small. Each kind of object is associated with a procedure that determines whether it should be subdivided further into other primitives or diced. An object is subdivided further if no method has been provided for dicing it directly, if it is determined that it would give rise to too many micropolygons, or if it is estimated that its micropolygons would differ too much in their final projected size. This recursive subdivision must ultimately result in objects that can be diced. To avoid the need to clip objects analytically to the view volume, when Reyes subdivides objects, only those parts that are at least partially within the view volume are kept. Perspective problems that would result from projecting an object that is too close to or behind the eye are avoided by subdividing further any object that spans both the hither plane and another plane that lies slightly in front of the eye.

Dicing an object results in a quadrilateral mesh of micropolygons that is shaded in WC. Because the micropolygons are sufficiently small, each is given a single shade, avoiding all the interpolated shading problems discussed in Section 16.2.6. Since a patch is diced parallel to its (s,t) coordinate system, some of the texture-mapping approaches discussed in Chapter 17 are particularly efficient to use. Dicing and shading can both take advantage of incremental algorithms. Reyes relies on the mapping techniques discussed in this chapter for its global lighting effects.

Visible surface determination is done with a subpixel z-buffer whose subpixel centers are jittered to accomplish stochastic sampling. The closest micropolygon covering a subpixel center is visible at that subpixel. To avoid the need to store micropolygon meshes and subpixel z and intensity values for the entire image, Reyes uses spatial partitioning. The image is divided into rectangular partitions into which each object is sorted by the upper left hand corner of its extent. The partitions are then processed left to right, top to bottom. As objects are subdivided or diced, the resulting subobjects or micropolygons are placed in the partitions that they intersect. Thus, only enough z-buffer memory is needed for a single partition, and other storage needed for a partition can be freed after it is processed.

16.14.4 Progressive Refinement

One interesting modification to the pipelines that we have discussed takes advantage of the fact that the image is viewed for a finite time. Instead of attempting to render a final version of a picture all at once, we can first render the picture coarsely, and then *progressively refine* it, to improve it. For example, a first image might have no antialiasing, simpler object models, and simpler shading. As the user views an image, idle cycles may be spent improving its quality [FORR85]. If there is some metric by which to determine what to do next, then refinement can occur adaptively. Bergman, Fuchs, Grant, and Spach [BERG86b] have developed such a system that uses a variety of heuristics to determine how it should spend its time. For example, a polygon is Gouraud-shaded, rather than constant-shaded, only if the range of its vertex intensities exceeds a threshold. Ray-tracing [PAIN89] and radiosity [COHE88] algorithms are both amenable to progressive refinement.

16.15 SUMMARY

In this chapter, we encountered many different illumination models, some inspired primarily by the need for efficiency, others that attempt to account for the physics of how surfaces actually interact with light. We saw how interpolation could be used in shading models, both to minimize the number of points at which the illumination equation is evaluated, and to allow curved surfaces to be approximated by polygonal meshes. We contrasted local illumination approaches that consider in isolation each surface point and the lights illuminating each point directly, with global approaches that support refraction and reflection of other objects in the environment. In each case, we noted that there are some methods that use the full geometric description of the environment in computing global effects, and others that use simpler descriptions, such as reflection maps.

As we have stressed throughout this chapter, the wide range of illumination and shading algorithms gives rise to a corresponding diversity in the images that can be produced of the same scene with the same viewing specification. The decision about which algorithms should be used depends on many factors, including the purposes for which an image is to be rendered. Although photorealism is often sacrificed in return for efficiency, advances in algorithms and hardware will soon make real-time implementations of physically correct, global illumination models a reality. When efficiency is no longer an issue, however, we may still choose to render some images without texture, shadows, reflections, or refraction, because in some cases this will remain the best way to communicate the desired information to the viewer.

EXERCISES

16.1 (a) Describe the difference in appearance you would expect between a Phong illumination model that used $(\overline{N} \cdot \overline{H})^n$ and one that used $(\overline{R} \cdot \overline{V})^n$. (b) Show that $\alpha = 2\beta$ when all vectors of Fig. 16.12 are coplanar. (c) Show that this relationship is *not* true in general.

16.2 Prove that the results of interpolating vertex information across a polygon's edges and scan lines are independent of orientation in the case of triangles.

16.3 Suppose there are polygons A, B, and C intersecting the same projector in order of increasing distance from the viewer. Show that, in general, if polygons A and B are transparent, the color computed for a pixel in the intersection of their projections will depend on whether Eq. (16.25) is evaluated with polygons A and B treated as polygons 1 and 2 or as polygons 2 and 1.

16.4 Consider the use of texture mapping to modify or replace different material properties. List the effects you can produce by mapping properties singly or in combination. How would you apply antialiasing to them?

16.5 Although using a reflection map may appear to require precomputing the lighting for the environment, a reflection map containing object identities and surface normals could be used instead. What are the disadvantages of using this kind of map?

16.6 Explain how to simulate reflections from surfaces of different roughness using a reflection map.

16.7 What other lighting effects can you think of that would generalize Warn's flaps and cones?

16.8 Suppose that the array of patches shown in Fig. 16.64 is continued for another two rows, adding patches 5 and 6, and that the radiosity values for the patches are $B_1 = B_2 = 2$, $B_3 = B_4 = 4$, $B_5 = B_6 = 6$. Show that B_k and B_r are 5 and 3, respectively. Then show that B_b is 1. Is this a reasonable value?

Notice that it extends the linear trend from h to e. What happens as you add more rows of patches in a similar pattern? Suppose that you added a mirror image of the patches about the line ac and computed the radiosity values. Then B_b would be 2. Does this seem contradictory? Explain your answer.

16.9 Implement a simple recursive ray tracer based on the material in Sections 15.10 and 16.12.

16.10 Make your ray tracer from Exercise 16.9 more efficient by using some of the techniques discussed in Section 16.12.1.

16.11 Extend your ray tracer from Exercise 16.10 to do distributed ray tracing.

16.12 Implement a progressive-refinement radiosity algorithm, based on the pseudocode of Fig. 16.70. Use the hemicube method of computing form factors. Begin by computing only patch to patch exchange (ignoring substructuring). Leave out the ambient computation to make coding and visual debugging easier. Check your hemicube code by verifying that the delta form factors sum to (approximately) 1.

To display your images, you will need to implement a polygon visible-surface algorithm (perhaps the one used by your hemicube) or have access to an existing graphics system. Using constant-shaded polygons will improve interactivity if shaded graphics hardware is not available (and will make programming and debugging easier).

16.13 Explain why lighting must be done before clipping in the pipeline of Fig. 16.72.

16.14 Implement a testbed for experimenting with local illumination models. Store an image that contains for each pixel its visible surface's index into a table of material properties, the surface normal, the distance from the viewer, and the distance from and normalized vector to one or more light sources. Allow the user to modify the illumination equation, the intensity and color of the lights, and the surface properties. Each time a change is made, render the surface. Use Eq. (16.20) with light-source attenuation (Eq. 16.8) and depth-cueing (Eq. 16.11).

16.15 Add a shadow algorithm to a visible-surface algorithm that you have already implemented. For example, if you have built a z-buffer system, you might want to add the two-pass z-buffer shadow algorithm discussed in Section 16.4.4. (The postprocessing variant may be particularly easy to add if you have access to a graphics system that uses a hardware z-buffer. Explain how extra storage at each pixel, as described in Exercise 16.14, could be used to design a shadow postprocess that produced correct shading and proper highlights.)

16.16 Add interobject reflections to a visible-surface algorithm. Use reflection mapping for curved surfaces and the mirror approach for planar surfaces, as described in Section 16.6.

18

Advanced Raster Graphics Architecture

Steven Molnar
and Henry Fuchs

In this chapter, we discuss in more detail the issues of raster graphics systems architecture introduced in Chapter 4. We examine the major computations performed by raster systems, and the techniques that can be used to accelerate them. Although the range of graphics architectures and algorithms is wide, we concentrate here on architectures for displaying 3D polygonal models, since this is the current focus in high-end systems and is the foundation for most systems that support more complex primitives or rendering methods.

Graphics systems architecture is a specialized branch of computer architecture. It is driven, therefore, by the same advances in semiconductor technology that have driven general-purpose computer architecture over the last several decades. Many of the same speed-up techniques can be used, including pipelining, parallelism, and tradeoffs between memory and computation. The graphics application, however, imposes special demands and makes available new opportunities. For example, since image display generally involves a large number of repetitive calculations, it can more easily exploit massive parallelism than can general-purpose computations. In high-performance graphics systems, the number of computations usually exceeds the capabilities of a single CPU, so parallel systems have become the rule in recent years. The organization of these parallel systems is a major focus of graphics architecture and of this chapter.

We begin by reviewing the simple raster-display architecture described in Chapter 4. We then describe a succession of techniques to add performance to the system, discussing the bottlenecks that arise at each performance level and techniques that can be used to overcome them. We shall see that three major performance bottlenecks consistently resist attempts to increase rendering speed: the number of floating-point operations to perform

ATI Ex. 2003
IPR2023-00564
Page 174 of 241

geometry calculations, the number of integer operations to compute pixel values, and the number of frame-buffer memory accesses to store the image and to determine visible surfaces. These demands have a pervasive influence on graphics architecture and give rise to the diversity of multiprocessor graphics architectures seen today. At the end of the chapter, we briefly discuss several unusual architectures, such as those for ray tracing, for true 3D displays, and for commercial flight simulators.

18.1 SIMPLE RASTER-DISPLAY SYSTEM

As described in Chapter 4, a simple raster-display system contains a CPU, system bus, main memory, frame buffer, video controller, and CRT display (see Fig. 4.18). In such a system, the CPU performs all the modeling, transformation, and display computations, and writes the final image to the frame buffer. The video controller reads pixel data from the frame buffer in raster-scan order, converts digital pixel values to analog, and drives the display.

It is important to remember that, if such a system has sufficient frame-buffer memory, has a suitable CRT display, and is given enough time, it can generate and display scenes of virtually unlimited complexity and realism. None of the architectures or architectural techniques discussed here enhance this fundamental capability (except for a few exotic 3D displays treated in Section 18.11). Rather, most work in graphics architecture concerns the quest for increased rendering speed.

In Chapter 4, we discussed two problems that limit the performance of this simple system: the large number of frame-buffer memory cycles needed for video scanout and the burden that image generation places on the main CPU. We now consider each of these problems in greater detail.

18.1.1 The Frame-Buffer Memory-Access Problem

In Chapter 4, we calculated the time between successive memory accesses when a low-resolution monochrome display is being refreshed. For a system with 16-bit words, the access rate is substantial—one memory access every 864 nanoseconds. Systems with higher-resolution color monitors require much higher memory speeds. For example, refreshing a 1280 by 1024 screen with 32-bit (one-word) pixels at 60 Hz requires that memory accesses occur every $1/(1280 \cdot 1024 \cdot 60) = 12.7$ nanoseconds. Even this is only the average memory access rate, not the peak rate, since pixels are not scanned out during horizontal and vertical retrace times [WHIT84]. A simple *dynamic random-access memory* (DRAM) system, on the other hand, has a cycle time of approximately 200 nanoseconds, a factor of 16 slower than the speed required. Clearly, something must be done to increase the bandwidth to frame-buffer memory.

The following sections discuss solutions that have been used by various system designers. Some of these provide only modest performance increases, but are sufficient for low-resolution systems. Others provide greatly increased memory bandwidth, but incur significant system complexity. We begin by reviewing briefly the fundamentals of DRAM

memories, since the characteristics of DRAMs strongly influence the set of solutions available.

18.1.2 Dynamic Memories

DRAMs are the memories of choice for most computer memory systems. *Static random-access memories* (SRAMs), which retain stored data indefinitely, can be made to run faster than DRAMs, which must be accessed every few milliseconds to retain data, but DRAMs are much denser and cheaper per bit. (DRAMs also require more complicated timing for reading and writing data, but these problems are easily solved with supporting circuitry.)

Figure 18.1 is a block diagram of a typical 1-Mbit DRAM chip. As in most DRAMs, single-bit storage elements are arranged in one or more square arrays (in this case, four arrays, each with dimension 512 by 512). Vertical *bit lines* transfer data to and from the storage arrays, one bit line for each column of each array. During read and write operations, one memory cell in each column is connected to its corresponding bit line. A *sense amplifier* attached to each bit line amplifies and restores the tiny signals placed on the bit line during read operations.

In a DRAM chip, read and write operations each require two steps. The first step is to select a row. This is done by asserting the *row address strobe* (RAS) while the desired row address is on the address inputs. The row decoder produces a 512-bit vector, whose bits are

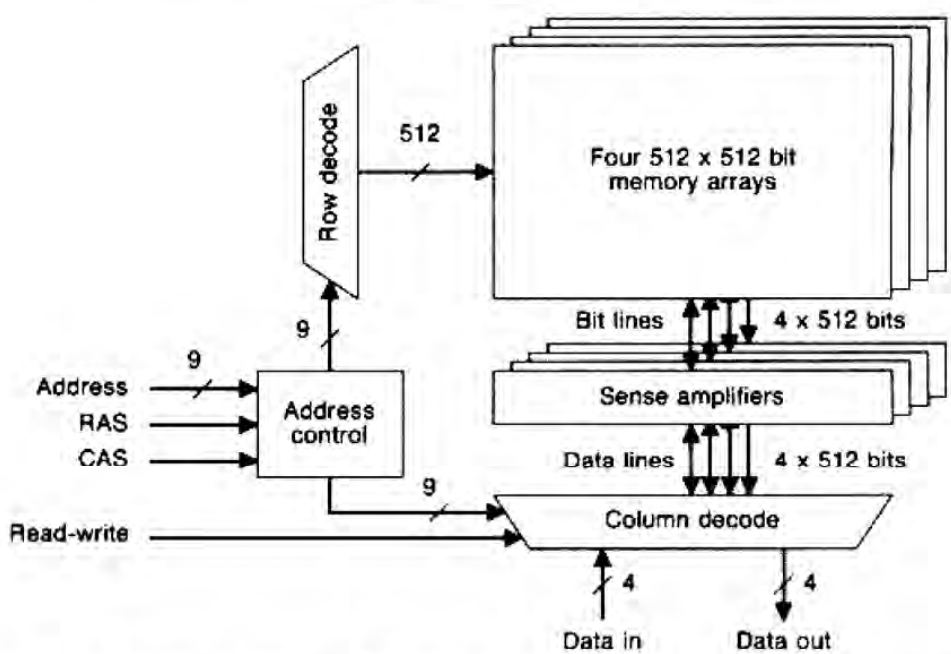


Fig. 18.1 A 1-Mbit (256K × 4) DRAM chip. An entire row is written to or read from each of the four memory arrays at the same time. The column decoder allows a particular element (column) of the selected row to be accessed.

0 everywhere except for a single 1 at the selected row. This bit vector determines which row's storage cells are connected to the bit lines and sense amplifiers.

The second step is to select a column, which is done by asserting the *column address strobe* (CAS) and read–write signal while the desired column address is on the address inputs. The column address selects a single bit from the active row of memory in each array. The selected bits are either buffered for output (during read operations) or set to the value on the data inputs (during write operations). Some DRAMs provide a faster access method called *page mode* for successive reads or writes to memory locations on the same row. In page mode, a row is selected just once, and successive columns are selected using the column address bits and CAS signal. In page mode, consecutive memory accesses require just one address cycle, instead of two. Thus, if adjacent pixels are stored in the same memory row, page mode can nearly double the bandwidth available for updating and displaying from the frame buffer.

Word widths greater than the number of data pins on a single memory chip can be accommodated by connecting multiple memory chips in parallel. With this arrangement, an entire word of data can be read from or written to the memory system in a single memory cycle. For example, eight 4-bit-wide DRAMs can be used to build a 32-bit memory system.

18.1.3 Increasing Frame-Buffer Memory Bandwidth

As we have seen, obtaining sufficient frame-buffer memory bandwidth both for video scanout and for updates by the CPU is a daunting problem. Different solutions have been proposed and used in various systems.

A partial solution, suitable for low-resolution displays, is to place the frame buffer on an isolated bus. This allows video scanout to occur simultaneously with CPU operations that do not affect the frame buffer. Only when the CPU needs to read from or write to the frame buffer must it contend with video scanout for memory cycles.

A second partial solution is to take advantage of DRAMs' page-mode feature. Since video scanout accesses memory locations in order, DRAM page misses are infrequent, so the frame buffer can run at almost twice its normal speed. Note, however, that frame-buffer accesses by the CPU may be less regular and may not be able to take advantage of page mode. We can decrease the number of page misses by the CPU by providing it with a data cache. As in conventional computing, a cache decreases the traffic between the CPU and memory, since a memory location accessed during one cycle is very likely to be accessed again in the near future. Furthermore, since cache entries frequently occupy several words, transferring data between the cache and main memory can exploit page mode. By combining page mode and a data cache, we can reduce the memory-access problem substantially, although further measures, as discussed in Section 18.1.5, are necessary for high-resolution displays.

A third, independent approach is to duplicate the frame-buffer memory, creating a *double-buffered* system in which the image in one buffer is displayed while the image in the other buffer is computed. Figure 18.2 shows one possible implementation, in which multiplexers connect each frame buffer to the system bus and video controller. Double-buffering allows the CPU to have uninterrupted access to one of the buffers while the video

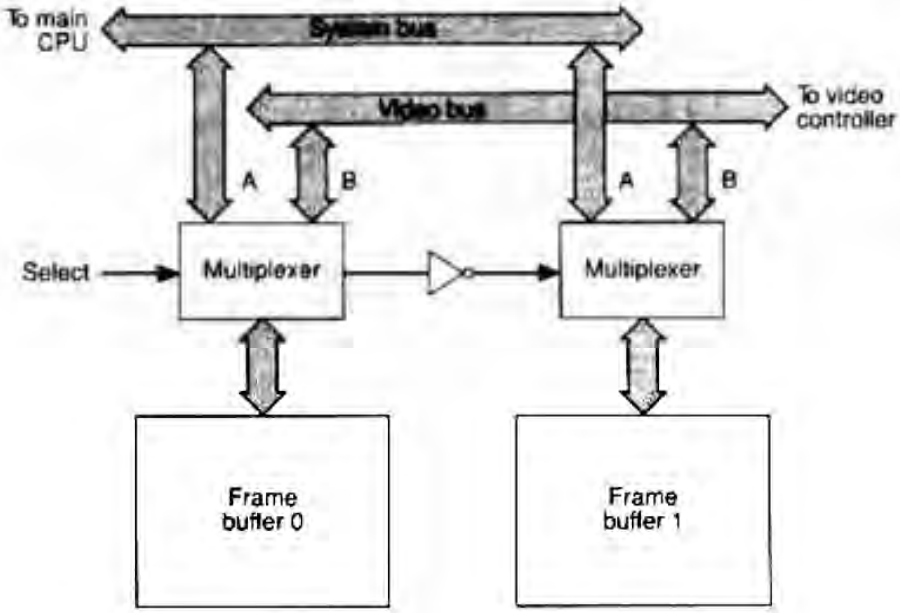


Fig. 18.2 Double-buffered frame buffer.

controller has uninterrupted access to the other. Double-buffering in this manner is expensive, however, since twice as much memory is needed as for a single-buffered display. Also, the multiplexers that provide dual access to the DRAMs require numerous chips, which increase the size of the system. (Actual dual-ported memories, with two fully independent read–write ports, could be used, but their low density and high cost make them impractical in almost all applications.)

18.1.4 The Video RAM

In 1983, Texas Instruments introduced a new type of DRAM, the *video random-access memory* (VRAM), designed specifically to allow video scanout to be independent of other frame-buffer operations [PINK83]. A VRAM chip, as shown in Fig. 18.3, is similar to a conventional DRAM chip, but contains a parallel-in/serial-out data register connected to a second data port. The serial register is as wide as the memory array and can be parallel loaded by asserting the transfer signal while a row of memory is being read. The serial register has its own data clock, enabling it to transfer data out of the chip at high speeds. The serial register and port effectively provide a second, serial port to the memory array. If this port is used for video scanout, scanout can occur in parallel with normal reads from and writes to the chip, virtually eliminating the video-scanout problem.

Since the shift register occupies only a small fraction of a VRAM's chip area and very few pins are needed to control it, VRAMs ideally should be only slightly more expensive than DRAMs. Unfortunately, the economics of scale tend to raise the price of VRAMs relative to DRAMs, since fewer VRAMs are produced (in 1989, VRAMs were roughly twice as expensive as DRAMs of the same density). In spite of this price differential, VRAMs are an excellent choice for many frame-buffer memories.

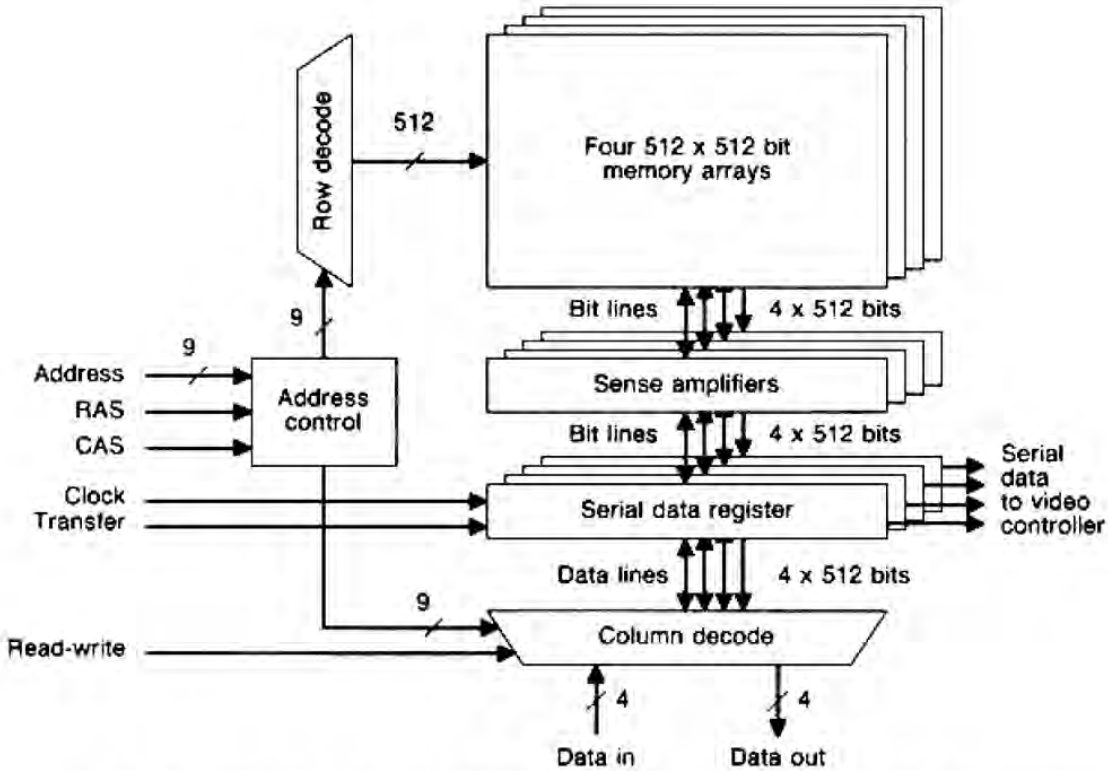


Fig. 18.3 Diagram of a 1-Mbit (256K × 4) VRAM chip. The serial data register provides a second (serial) port to the memory array

18.1.5 A Frame Buffer for a High-Resolution Display

Figure 18.4 shows a typical frame-buffer design using VRAMs. The CPU has essentially unrestricted access to the frame-buffer memory. Only once every *n* cycles (where *n* is the dimension of the memory array) is a memory cycle required to load a new row into the VRAMs' serial registers. With screen resolutions up to 512 by 512 pixels, VRAMs can handle video scanout unassisted. With higher-resolution displays, the pixel rate exceeds the VRAMs' serial-port data rate of 30 MHz or less. A standard solution is to connect multiple banks of VRAMs to a high-speed off-chip shift register, as shown in the figure. Multiple pixels, one from each VRAM bank, can be loaded into the shift register in parallel and then shifted out serially at video rates. (Notice that this 1280- by 1024-pixel system, with five times the resolution of a 512 by 512 system, uses five-way multiplexing.) This shift register frequently is incorporated into a single chip containing look-up tables and digital-to-analog converters, such as Brooktree's Bt458 RAMDAC [BROO88]. In systems with 24 bits per pixel, such as the one shown in Fig. 18.4, three RAMDAC chips are typically used—one for each 8-bit color channel.

VRAMs provide an elegant solution to the frame-buffer memory-access problem. The bottleneck that remains in such systems is CPU processing speed. To generate a raster image (in 2D or 3D), the processor must calculate every pixel of every primitive. Since a typical primitive covers many pixels, and images typically contain thousands of primitives,

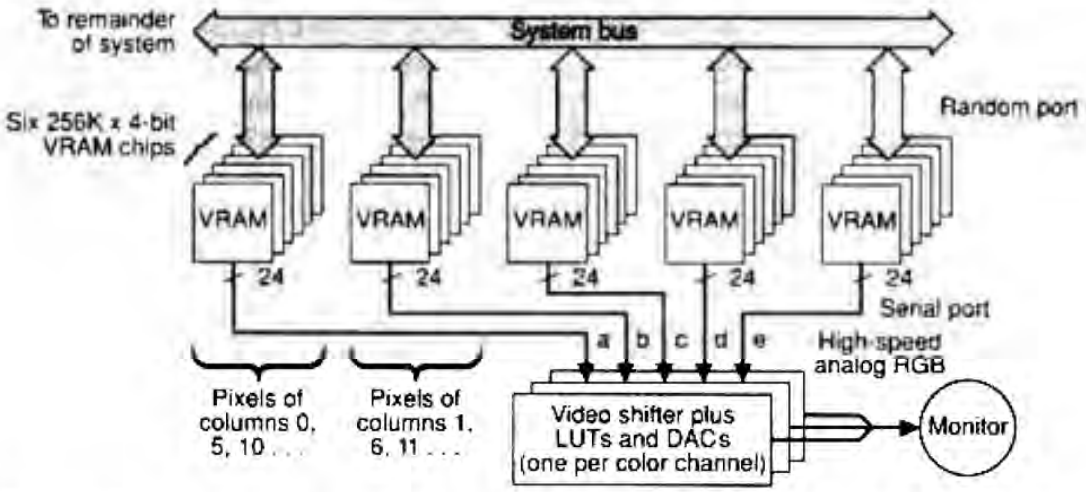


Fig. 18.4 Block diagram of a 1280- by 1024-pixel display using VRAMs. A parallel-in/serial-out shift register provides the high pixel rates required to drive the monitor.

generating a high-resolution image can require many millions of CPU operations. Because of this, even the fastest CPUs have difficulty rendering images quickly enough for interactive use.

18.2 DISPLAY-PROCESSOR SYSTEMS

To build a system with higher performance than the single-CPU system described in the previous section, we must either remove some of the graphics burden from the CPU or increase the CPU's ability to perform these tasks. Here we consider strategies that do not substantially increase the size or cost of the system (we consider larger, multiprocessing solutions in Sections 18.6 through 18.10). These strategies generally fall into three categories: (1) coprocessors that share the system bus with the main CPU, (2) display processors with their own buses and memory systems, and (3) integrated processors containing internal hardware support for graphics operations.

Coprocessors were discussed extensively in Chapter 4 (single-address-space (SAS) architectures). These have been found useful in 2D systems, but have not achieved wide use in 3D systems, largely because 3D systems require a larger set of operations than 2D systems and because bus contention in a SAS architecture severely limits system performance. The latter two approaches have been used successfully for 3D graphics, and we now discuss them further.

18.2.1 Peripheral Display Processors

Figure 4.22 shows a typical peripheral-display-processor system organization. The display processor generally has its own memory system (including the frame buffer) and a fast communication link to the main CPU. The main CPU sends display commands to the display processor, which executes them using resident software or firmware routines, or using specialized hardware.

The display processor may be the same processor type as the main CPU, or it may be a special-purpose processor. Using the same type of processor, as in the Masscomp MC-500 [MASS85], allows simpler and more flexible hardware and software designs. It does not exploit the regularity and specialized nature of most graphics calculations, however, and therefore provides at most a factor of 2 increase in performance, since the overall processing capacity is simply doubled. Most current display-processor systems use peripheral processors specialized for graphics operations.

Single-chip graphics processors. Single-chip graphics processors are a successful and inexpensive type of peripheral-display processor that developed as functional extensions of single-chip video controllers. Consequently, many current graphics-processor chips combine the functions of the video controller and display processor. (Section 18.2.2 describes one popular graphics processor chip, the TMS34020.)

Graphics-processor chips targeted especially for 3D graphics have been introduced only recently. A pioneering chip of this type is Intel's i860, which can be used either as a stand-alone processor with support for graphics or as a 3D graphics processor. Unlike many 2D graphics-processor chips, the i860 does not provide hardware support for screen refresh or video timing. The i860 will be discussed in depth in Section 18.2.4.

Programming the display processor. A major point of difference among display processor systems is where they store the model or database from which the image is generated: in the CPU's memory, or in the display processor's memory. Storing the database in the display processor's memory frees the main CPU from database traversal. Consequently, only a low-bandwidth channel is needed between the CPU and the display processor. However, such a system also inherits all the disadvantages of a structure database described in Chapter 7, particularly the inflexibility of a canonical, "hard-wired" database model.

Whether or not the display processor manages the database, it is an extra processor to program. Frequently, this program takes the form of a standard graphics library, such as PHIGS+, which satisfies the needs of many applications. Applications invariably arise, however, that require features not supported by the library. In such cases, users are forced either to program the display processor themselves (which may not even be possible), or to devise cumbersome workarounds using existing display-processor features.

18.2.2 Texas Instruments' TMS34020—A Single-Chip Peripheral Display Processor[1]

Texas Instruments' TMS34020 [TEXA89] is an example of a single-chip graphics processor designed to accelerate 2D displays on PCs and workstations. It can be paired with a TMS34082 floating-point coprocessor, which rapidly performs 3D geometric transformations and clipping needed for 3D graphics. Unlike graphics-processor chips that perform only graphics-specific operations, the 34020 is a fully programmable 32-bit processor that can be used as a stand-alone processor if desired.

[1]Material for this example was contributed by Jerry R. Van Aken of Texas Instruments, Inc.

Figure 18.5 shows a typical system configuration using the 34020 (and optionally the 34082). As indicated in the figure, the graphics processor's local memory may contain VRAMs for the frame buffer and DRAMs for code and data storage. Alternatively, the graphics processor's entire memory may be built from VRAMs. The 34020 contains on-chip timers and registers to control video scanout and DRAM refreshing.

The 34020 supports data types for pixels and pixmaps, allowing pixels to be accessed by their (x, y) coordinates as well as by their addresses in memory. The following instructions implement common 2D graphics operations:

PIXBLT: The PIXBLT (pixel-block transfer) instruction implements a general bitBlt operation. It copies pixels of 1 to 8 bits, automatically aligning, masking, and clipping rows of pixels fetched from the source array before writing them to the destination array. It uses many of the optimizations discussed in Chapter 19, as well as processing several pixels in parallel, so the transfer rate approaches the system's full memory bandwidth—up to 18 million 8-bit pixels per second. A special version of PIXBLT supports text by efficiently expanding packed 1-bit bitmaps into 8-bit pixmaps.

FILL: The FILL instruction fills a rectangular array of pixels with a solid color. Large areas are filled at rates approaching the memory-bus speed. Texas Instruments' 1-Mbit VRAM supports a special block-write mode that allows up to four memory locations to be written at once from the same on-chip color register. Using this block-write mode, the 34020 fills areas at up to 160 million 8-bit pixels per second.

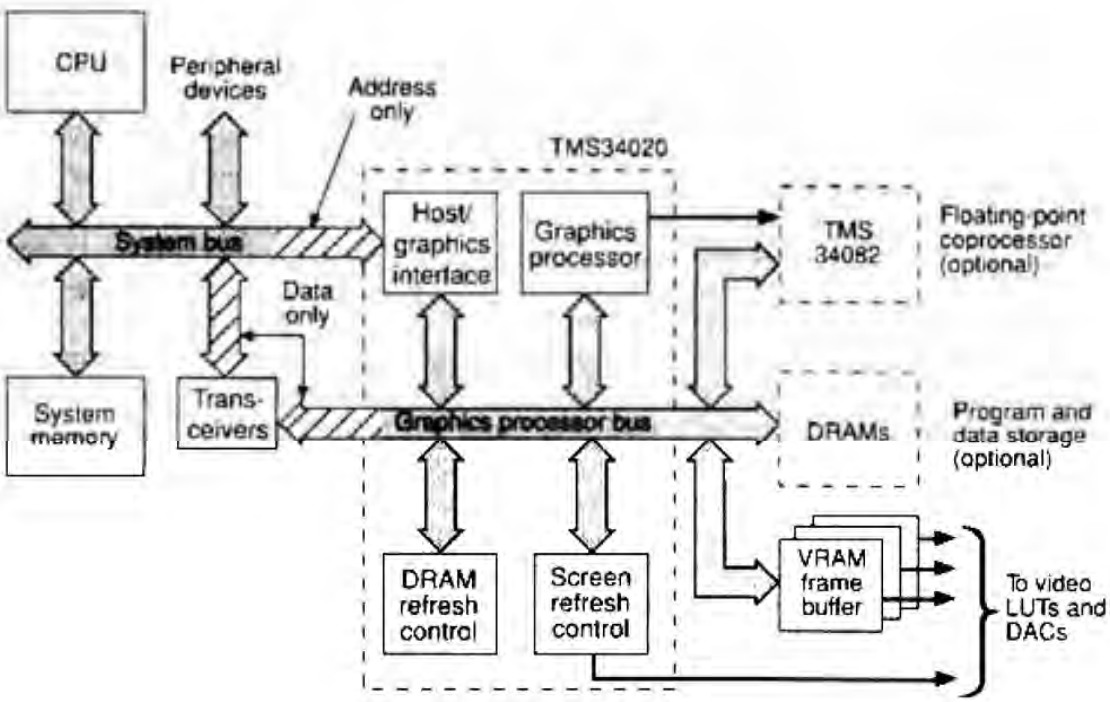


Fig. 18.5 Block diagram of the TMS34020 graphics-processor chip used as a peripheral display processor.

FLINE: The FLINE instruction draws 1-pixel-thick straight lines using a midpoint scan-conversion algorithm (see Section 3.2.2). The FLINE instruction draws up to 5 million pixels per second.

DRAV: The DRAV (draw-and-advance) instruction draws the pixel indicated by a pair of (x, y) coordinates, then increments the x and y coordinates in parallel. It is used in the inner loops of incremental algorithms for drawing circles and ellipses; for example, the inner loop of a midpoint circle routine (see Section 3.3.2) draws 2 million pixels per second.

Software for the 34020 is typically divided into time-critical graphics subroutines, written in assembly language, and other software, written in a high-level language. Software running on the 34020 communicates with the application program running on the main CPU.

The 34020 contains a 512-byte on-chip instruction cache and 30 general-purpose registers—enough memory to store the instructions and data required within the inner loop of many graphics subroutines. Once the instructions and data for the loop have been read from external memory, the memory bus is available exclusively for moving pixels to and from the frame buffer.

The TMS34082 floating-point coprocessor chip interfaces directly to the 34020, providing floating-point instructions and registers that increase the system's capabilities for 3D and other floating-point-intensive applications. The 34082 monitors the 34020's memory bus and receives floating-point instructions and data transmitted over the bus by the 34020. The 34082 can transform a 3D homogeneous point (including the homogeneous divide) every 2.9 microseconds.

18.2.3 Integrated Graphics Processors

Another approach to increasing system performance is to add support for graphics directly to the main CPU. This has become feasible only recently, with the development of very dense VLSI technologies that allow entire systems to be implemented on a single chip. Fully programmable 2D graphics processors, such as TI's 34020, are early examples of this approach, although they dedicate much of their chip area to graphics functions and comparatively little to general-purpose processing.

Intel's i860 is the first microprocessor chip to integrate direct support for 3D graphics. Proponents of this approach argue that the chip area required to support reasonable 3D graphics performance is relatively small and the payoff is high, since 3D graphics has become an integral part of many computing environments. This approach may well produce high-performance, low-cost 3D systems as increasing VLSI densities allow more sophisticated systems and more system components to be built onto a single chip.

18.2.4 Intel's i860—A Single-Chip Microprocessor with Integrated Support for 3D Graphics

Advances in VSLI technology have made it possible to build chips with extremely large numbers of transistors—more than 1 million in 1989. This allows high-performance CPU chips to include other features, such as caches, I/O controllers, and support for specialized

instructions. Intel's i860 microprocessor, also known as the 80860, [GRIM89] is an example of such a chip with features supporting 3D graphics. Figure 18.6 is a block diagram of the major data paths in the i860.

According to Intel, the i860 achieves the following benchmarks: 33 million VAX-equivalent instructions per second (on compiled C code), 13 double-precision MFLOPs (using the Linpack benchmark), and 500,000 homogeneous vector transformations [INTE89].

The i860's graphics instructions operate in parallel on as many pixels as can be packed into a 64-bit data word. For applications using 8-bit pixels, this means eight operations can occur simultaneously. For 3D shaded graphics, which generally requires 32-bit pixels, two operations can occur simultaneously. Parallel graphics instructions include:

- Calculating multiple linear interpolations in parallel
- Calculating multiple z-buffer comparisons in parallel
- Conditionally updating multiple pixels in parallel.

For systems using 8-bit pixels (where eight operations can occur in parallel), the i860 can scan convert and shade 50,000 Gouraud-shaded 100-pixel triangles per second.

The i860 may be used either as a peripheral display processor (with an 80486

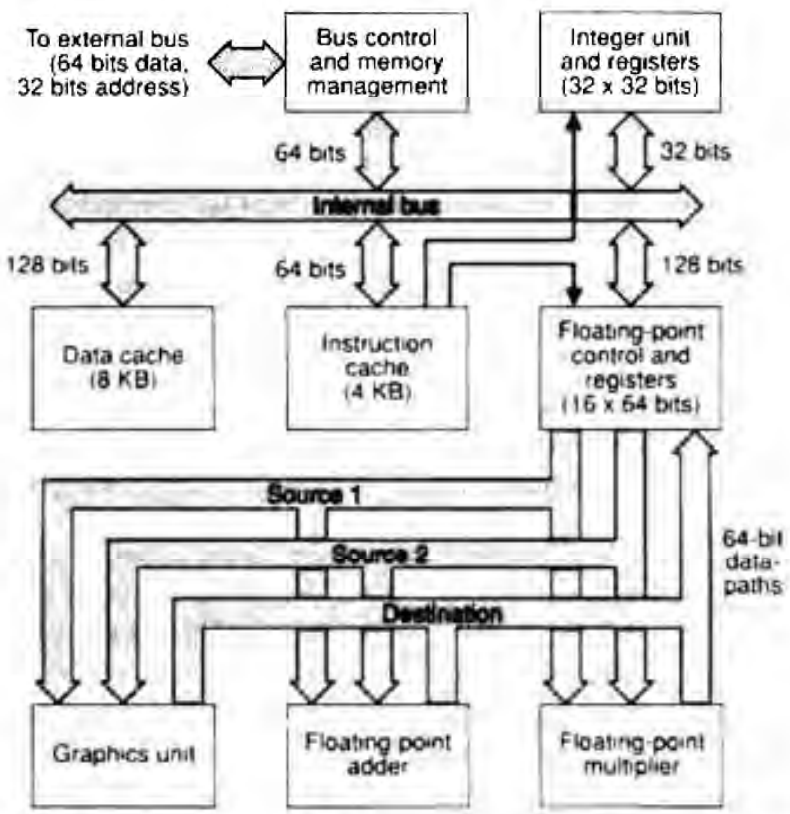


Fig. 18.6 Block diagram of Intel i860 microprocessor datapaths (adapted from [GRIM89])

microprocessor as the main CPU, for example) or as a stand-alone processor. By combining front-end and back-end capabilities in a single processor, the i860 allows a powerful 3D graphics system to be built with very few parts—essentially the processor, memory, and the video system.

As VLSI densities continue to improve, we can expect to see even higher performance and more complex processors designed for graphics. Features commonly found in 2D graphics-processor chips, such as scanout and video-timing circuitry, may be included in such processors as well.

18.2.5 Three Performance Barriers

Systems can be built with a large range of performances using the techniques described up to this point. Low-end systems using a general-purpose microprocessor for the application processor and a 2D graphics processor are well suited for displays on PCs. Higher-performance systems using a powerful application processor and a high-speed 3D graphics processor, containing display-list memory and transformation hardware, are suitable for engineering and scientific workstations.

But how far can such designs be pushed? Let us imagine a display system made with the fastest available CPU and display processor, and a frame buffer containing the fastest available VRAMs. Such a system will achieve impressive—but still limited—performance. In practice, such *single-stream* architectures have been unable to achieve the speeds necessary to display large 3D databases at interactive rates. Architectures that display more primitives at faster update rates must overcome three barriers:

- *Floating-point geometry processing*: Accelerating the transformation and clipping of primitives beyond the rates possible in a single floating-point processor
- *Integer pixel processing*: Accelerating scan conversion and pixel processing beyond the rates possible in a single display processor and memory system
- *Frame-buffer memory bandwidth*: Providing higher bandwidth into the frame buffer (faster reads and writes) than can be supported by a conventional memory system.

These barriers have analogs in conventional computer design: the familiar processing and memory-bandwidth bottlenecks. In conventional computers, the presence of these bottlenecks has led to the development of concurrent processing. Similarly, the major focus of research in graphics architecture over the last decade has been investigating ways to use concurrency to overcome these performance barriers.

Before we discuss parallel graphics architectures, let us establish some groundwork about the demands of graphics applications and the fundamentals of multiprocessing. Later, we shall discuss various methods for building parallel systems that overcome these three performance barriers.

18.3 STANDARD GRAPHICS PIPELINE

Here we review from a hardware perspective the standard graphics pipeline discussed in Chapter 16. As has been mentioned, the rendering pipeline is a logical model for the computations needed in a raster-display system, but is not necessarily a physical model, since the stages of the pipeline can be implemented in either software or hardware.

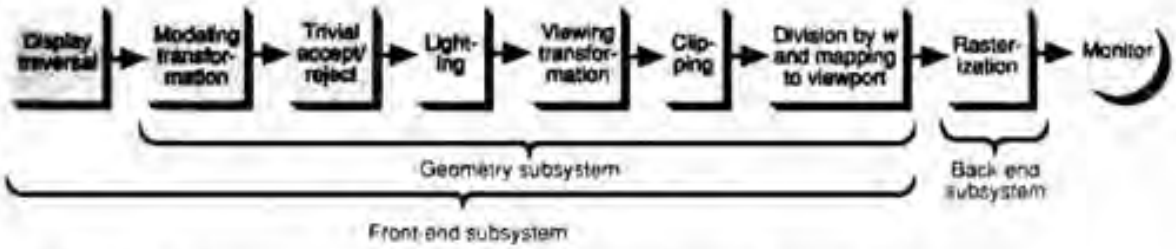


Fig. 18.7 Standard graphics pipeline for Gouraud- or Phong-shaded polygons.

Figure 18.7 shows a version of the rendering pipeline that is typical for systems using conventional primitives (lines and polygons) and conventional shading techniques (constant, Gouraud, or Phong). We shall discuss each stage of the pipeline in turn, paying particular attention to algorithms that have been used successfully in hardware systems and to those that may be useful in the future. At the end of this section, we estimate the number of computations needed at each stage of the graphics pipeline to process a sample database containing 10,000 polygons.

18.3.1 Display Traversal

The first stage of the pipeline is traversal of the display model or database. This is necessary because the image may change by an arbitrary amount between successive frames. All the primitives in the database must be fed into the remainder of the display pipeline, along with context information, such as colors and current-transformation matrices. Chapter 7 described the two types of traversal: immediate mode and retained mode. Both methods have advantages and disadvantages, and the choice between them depends on the characteristics of the application and of the particular hardware architecture used.

Immediate mode offers flexibility, since the display model does not need to conform to any particular display-list structure and the application has the luxury of recreating the model differently for every frame. The main CPU must perform immediate-mode traversal, however, expending cycles it could use in other ways. Retained mode, on the other hand, can be handled by a display processor if the structure database is stored in its local memory. Retained-mode structure traversal can be accelerated by optimizing the database storage and access routines or by using a dedicated hardware traverser. Furthermore, since the main CPU only edits the database each frame, rather than rebuilding it from scratch, a low-bandwidth channel between the main CPU and the display processor is sufficient. Of course, relatively few changes can be made to the structure database between frames, or system performance will suffer.

The choice between traversal modes is a controversial matter for system designers [AKEL89]. Many argue that retained mode offers efficiency and high performance. Others believe that immediate mode supports a wider range of applications and does not necessarily lead to reduced performance if the system has a sufficiently powerful CPU.

Unfortunately, it is difficult to estimate the processing requirements for display traversal, since they depend on the traversal method used and on the characteristics of the particular display model. At the very least, a read operation and a write operation must be performed for each word of data to be displayed. The processing requirements may be much

greater if the structure hierarchy is deep or if it contains many modeling transformations (because of the overhead in pointer chasing, state saving, concatenating transformation matrices, etc.).

18.3.2 Modeling Transformation

In this stage of the pipeline, graphics primitives are transformed from the object-coordinate system to the world-coordinate system. This is done by transforming the vertices of each polygon with a single transformation matrix that is the concatenation of the individual modeling transformation matrices. In addition, one or more surface-normal vectors may need to be transformed, depending on the shading method to be applied.

Constant shading requires world-space surface-normal vectors for each polygon. We compute these by multiplying object-space surface normals by the transpose of the inverse modeling transformation matrix. Gouraud and Phong shading require world-space normals for each vertex, rather than for each polygon, so each vertex-normal vector must be multiplied by the transpose inverse transformation matrix.

Let us compute the number of floating-point calculations required to transform a single vertex if Gouraud shading is to be applied. Multiplying a homogeneous point by a 4×4 matrix requires 16 multiplications and 12 additions. Multiplying each vertex normal by the inverse transformation matrix requires 9 multiplications and 6 additions (only the upper-left 3×3 portion of the matrix is needed—see Exercise 18.1). Therefore, transforming a single vertex with surface normal requires $16 + 9 = 25$ multiplications and $12 + 6 = 18$ additions.

18.3.3 Trivial Accept/Reject Classification

In the trivial accept/reject classification stage, primitives (now in world coordinates) are tested to see whether they lie wholly inside or outside the view volume. By identifying primitives that lie outside the view volume early in the rendering pipeline, processing in later stages is minimized. We will clip primitives that cannot be trivially accepted or rejected in the clipping stage.

To trivially accept or reject a primitive, we must test each transformed vertex against the six bounding planes of the view volume. In general, the bounding planes will not be aligned with the coordinate axes. Each test of a vertex against a bounding plane requires 4 multiplications and 3 additions (the dot product of a homogeneous point with a 3D plane equation). A total of $6 \cdot 4 = 24$ multiplications and $6 \cdot 3 = 18$ additions are required per vertex.

18.3.4 Lighting

Depending on the shading algorithm to be applied (constant, Gouraud, or Phong), an illumination model must be evaluated at various locations: once per polygon for constant shading, once per vertex for Gouraud shading, or once per pixel for Phong shading. Ambient, diffuse, and specular illumination models are commonly used in high-performance systems.

In constant shading, a single color is computed for an entire polygon, based on the position of the light source and on the polygon's surface-normal vector and diffuse color (see Eq. 16.9). The first step is to compute the dot product of the surface-normal vector and the light vector (3 multiplications and 2 additions for directional light sources). If an attenuation factor based on the distance to the light source is used, we must calculate it and multiply it by the dot product here. Then, for each of the red, green, and blue color components, we multiply the dot product by the light-source intensity and diffuse-reflection coefficient (2 multiplications), multiply the ambient intensity by the ambient-reflection coefficient (1 multiplication), and add the results (1 addition). If we assume a single directional light source, calculating a single RGB triple requires $3 + 3 \cdot (2 + 1) = 12$ multiplications and $2 + 3 \cdot 1 = 5$ additions. Gouraud-shading a triangle requires three RGB triples—one for each vertex.

Phong shading implies evaluating the illumination model at each pixel, rather than once per polygon or once per polygon vertex. It therefore requires many more computations than does either constant or Gouraud shading. These computations, however, occur during the rasterization stage of the pipeline.

18.3.5 Viewing Transformation

In this stage, primitives in world coordinates are transformed to normalized projection (NPC) coordinates. This transformation can be performed by multiplying vertices in world coordinates by a single 4×4 matrix that combines the perspective transformation (if used) and any skewing or nonuniform scaling transformations needed to convert world coordinates to NPC coordinates. This requires 16 multiplications and 12 additions per vertex. Viewing transformation matrices, however, have certain terms that are always zero. If we take advantage of this, we can reduce the number of computations for this stage by perhaps 25 percent. We will assume that 12 multiplications and 9 additions per vertex are required in the viewing transformation stage.

Note that if a simple lighting model (one that does not require calculating the distance between the light source and primitive vertices) is used, modeling and viewing transformation matrices can be combined into a single matrix. In this case only one transformation stage is required in the display pipeline—a significant savings.

18.3.6 Clipping

In the clipping stage, lit primitives that were not trivially accepted or rejected are clipped to the view volume. As described in Chapter 6, clipping serves two purposes: preventing activity in one screen window from affecting pixels in other windows, and preventing mathematical overflow and underflow from primitives passing behind the eye point or at great distances.

Exact clipping is computationally practical only for simple primitives, such as lines and polygons. These primitives may be clipped using any of the 3D clipping algorithms described in Section 6.5. Complicated primitives, such as spheres and parametrically defined patches, are difficult to clip, since clipping can change the geometric nature of the primitive. Systems designed to display only triangles have a related problem, since a clipped triangle may have more than three vertices.

An alternative to exact clipping is scissoring, described in Section 3.11. Here, primitives that cross a clipping boundary are processed as usual until the rasterization stage, where only pixels inside the viewport window are written to the frame buffer. Scissoring is a source of inefficiency, however, since effort is expended on pixels outside the viewing window. Nevertheless, it is the only practical alternative for clipping many types of complex primitives.

In the pipeline described here, all clipping is performed in homogeneous coordinates. This is really only necessary for z clipping, since the w value is needed to recognize vertices that lie behind the eye. Many systems clip to x and y boundaries after the homogeneous divide for efficiency. This simplifies x and y clipping, but still allows primitives that pass behind the eye to be recognized and clipped before w information is lost.

The number of computations required for clipping depends on how many primitives cross the clipping boundaries, which may change from one frame to the next. A common assumption is that only a small percentage of primitives (10 percent or fewer) need clipping. If this assumption is violated, system performance may decrease dramatically.

18.3.7 Division by w and Mapping to 3D Viewport

Homogeneous points that have had a perspective transformation applied, in general, have w values not equal to 1. To compute true x, y, and z values, we must divide the x, y, and z components of each homogeneous point by w. This requires 3 divisions per vertex. In many systems, vertex x and y coordinates must be mapped from the clipping coordinate system to the coordinate system of the actual 3D viewport. This is a simple scaling and translation operation in x and y that requires 2 multiplications and 2 additions per vertex.

18.3.8 Rasterization

The rasterization stage converts transformed primitives into pixel values, and generally stores them in a frame buffer. As discussed in Section 7.12.1, rasterization consists of three subtasks: *scan conversion*, *visible-surface determination*, and *shading*. Rasterization, in principle, requires calculating each primitive's contribution to each pixel, an $O(nm)$ operation, where n is the number of primitives and m is the number of pixels.

In a software rendering system, rasterization can be performed in either of two orders: primitive by primitive (object order), or pixel by pixel (image order). The pseudocode in Fig. 18.8 describes each of these two approaches, which correspond to the two main families of image-precision visibility algorithms: z-buffer and scan-line algorithms.

Most systems today rasterize in object order, using the z-buffer algorithm to compute

```
for (each primitive P)
    for (each pixel q within P)
        update frame buffer based on color
            and visibility of P at q;
```
(a)

```
for (each pixel q)
    for (each primitive P covering q)
        compute P's contribution to q,
            outputting q when finished;
```
(b)

Fig. 18.8 Pseudocode for (a) object-order and (b) image-order rasterization algorithms.

visibility. The z-buffer algorithm has only recently become practical with the availability of inexpensive DRAM memory. If we assume that visibility is determined using a z-buffer, the number of computations required in the rasterization stage still depends on the scan-conversion and shading methods. Scan-conversion calculations are difficult to categorize, but can be significant in a real system.

Constant shading requires no additional computations in the rasterization stage, since the polygon's uniform color was determined in the lighting stage. Gouraud shading requires red, green, and blue values to be bilinearly interpolated across each polygon. Incremental addition can be used to calculate RGB values for each succeeding pixel. Phong shading requires significantly more computation. The x, y, and z components of the surface normal must be linearly interpolated across polygons based on the normal vectors at each vertex. Since the interpolated vector at each pixel does not, in general, have unit length, it must be normalized. This requires 2 additions, an expensive reciprocal square-root operation, and 3 multiplications for every pixel. Then, the Phong illumination model must be evaluated, requiring a dot product, more multiplications and additions, and, in the general case, the evaluation of an exponential. Bishop and Weimer's approximation to Phong shading, as mentioned in Section 16.2.5, can reduce some of these calculations, albeit at the expense of realism for surfaces with high specularity. Other shading techniques, such as transparency and texturing, may be desired as well. The architectural implications of these techniques are discussed in Section 18.11.

In addition to the shading calculations, updating each pixel requires reading the z-buffer, comparing old and new z values, and, if the primitive is visible, writing new color and z values to the frame buffer.

18.3.9 Performance Requirements of a Sample Application

In Section 18.2, we mentioned the three major performance barriers for raster graphics systems: the number of floating-point operations required in geometry calculations and the number of pixel-oriented computations and frame-buffer accesses required in rasterization. To appreciate the magnitude of these problems, we shall estimate the number of computations needed at each stage of the rendering pipeline to display a sample database at a modest update rate. To make this estimate, we must define a representative database and application.

A sample database. We use triangles for primitives in our sample database, since they are common and their computation requirements are easy to estimate (complex primitives with shared vertices, such as triangle strips and quadrilateral meshes, are becoming increasingly popular, however. Exercise 18.2 explores the savings that can be achieved using these primitives.) We assume a modest database size of 10,000 triangles, a size that current workstations can display at interactive rates [AKEL88; APGA88; BORD89; MEGA89]. In addition, we assume that each triangle covers an average of 100 pixels.

The number of polygons that fall on clipping boundaries can vary widely from one frame to the next. For simplicity, we assume that no primitives need clipping in our sample application. This means we underestimate the amount of calculations required for clipping, but maximize the work required in succeeding stages. Image *depth complexity*, the average number of polygons mapping to a pixel, depends on the database as well as on the view. We

assume arbitrarily that one-half of the pixels of all the triangles are obscured by some other triangle.

We assume that an ambient/diffuse illumination model and Gouraud shading are to be applied to each primitive (this is the lowest common standard for current 3D systems, although Phong illumination combined with Gouraud shading is becoming increasingly popular). We assume a screen size of 1280 by 1024 pixels and an update rate of 10 frames per second, typical of current interactive applications, but far from ideal.

In summary, our sample application has the following characteristics:

- 10,000 triangles (none clipped)
- Each triangle covers an average of 100 pixels, one-half being obscured by other triangles
- Ambient and diffuse illumination models (not Phong)
- Gouraud shading
- 1280 by 1024 display screen, updated at 10 frames per second.

We cannot calculate all the computation and memory-bandwidth requirements in our sample application, since many steps are difficult to categorize. Instead, we concentrate on the three performance barriers: the number of floating-point operations for geometry computations, the number of integer operations for computing pixel values, and the number of frame-buffer accesses for rasterization.

Geometry calculations. For each frame, we must process $10{,}000 \cdot 3 = 30{,}000$ vertices and vertex-normal vectors. In the modeling transformation stage, transforming a vertex (including transforming the normal vector) requires 25 multiplications and 18 additions. The requirements for this stage are thus $30{,}000 \cdot 25 = 750{,}000$ multiplications and $30{,}000 \cdot 18 = 540{,}000$ additions.

Trivial accept/reject classification requires testing each vertex of each primitive against the six bounding planes of the viewing volume, a total of 24 multiplications and 18 additions per vertex. The requirements for this stage are thus $30{,}000 \cdot 24 = 720{,}000$ multiplications and $30{,}000 \cdot 18 = 540{,}000$ additions, regardless of how many primitives are trivially accepted or rejected.

Lighting requires 12 multiplications and 5 additions per vertex, a total of $30{,}000 \cdot 12 = 360{,}000$ multiplications and $30{,}000 \cdot 5 = 150{,}000$ additions.

The viewing transformation requires 8 multiplications and 6 additions per vertex, a total of $30{,}000 \cdot 8 = 240{,}000$ multiplications and $30{,}000 \cdot 6 = 180{,}000$ additions.

The requirements for clipping are variable; the exact number depends on the number of primitives that cannot be trivially accepted or rejected, which in turn depends on the scene and on the viewing angle. We have assumed the simplest case for our database, that all primitives lie completely within the viewing volume. If a large fraction of the primitives needs clipping, the computational requirements could be substantial (perhaps even more than in the geometric transformation stage).

Division by w requires 3 divisions per vertex, a total of $30{,}000 \cdot 3 = 90{,}000$ divisions. Mapping to the 3D viewport requires 2 multiplications and 2 additions per vertex, a total of 60,000 multiplications and 60,000 additions.

Summing the floating-point requirements for all of the geometry stages gives a total of 2,220,000 multiplications/divisions and 1,470,000 additions/subtractions per frame. Since a new frame is calculated every $\frac{1}{10}$ second, a total of 22.2 million multiplications/divisions and 14.7 million additions/subtractions (36.9 million aggregate floating-point operations) as required per second—a very substantial number.

Rasterization calculations and frame-buffer accesses. Let us now estimate the number of pixel calculations and frame-buffer memory accesses required in each frame. We assume that z values and RGB triples each occupy one word (32 bits) of frame-buffer memory (typical in most current high-performance systems). For each pixel that is initially visible (i.e., results in an update to the frame buffer), z, R, G, and B values are calculated (4 additions per pixel if forward differences are used), a z value is read from the frame buffer (1 frame-buffer cycle), the z values are compared (1 subtraction) and new z values and colors are written (2 frame-buffer cycles). For each pixel that is initially not visible, only the z value needs to be calculated (1 addition), and a z value is read from the frame buffer (1 frame-buffer cycle), and the two z values are compared (1 subtraction). Note that initially visible pixels may get covered, but initially invisible pixels can never be exposed.

Since we assume that one-half of the pixels of each triangle are visible in the final scene, a reasonable guess is that three-quarters of the pixels are initially visible and one-quarter of the pixels are initially invisible. Each triangle covers 100 pixels, so $\frac{3}{4} \cdot 100 \cdot 10{,}000 = 750{,}000$ pixels are initially visible and $\frac{1}{4} \cdot 100 \cdot 10{,}000 = 250{,}000$ pixels are initially invisible. To display an entire frame, therefore, a total of $(750{,}000 \cdot 5) + (250{,}000 \cdot 2) = 4.25$ million additions and $(750{,}000 \cdot 3) + (250{,}000 \cdot 1) = 2.5$ million frame-buffer accesses is required. To initialize each frame, both color and z-buffers must be cleared, an additional $1280 \cdot 1024 \cdot 2 = 2.6$ million frame-buffer accesses. The total number of frame-buffer accesses per frame, therefore, is 2.5 million + 2.6 million = 5.1 million. If 10 frames are generated per second, 42.5 million additions and 51 million frame-buffer accesses are required per second.

In 1989, the fastest floating-point processors available computed approximately 20 million floating-point operations per second, the fastest integer processors computed approximately 40 million integer operations per second, and DRAM memory systems had cycle times of approximately 100 nanoseconds. The floating-point and integer requirements of our sample application, therefore, are just at the limit of what can be achieved in a single CPU. The number of frame-buffer accesses, however, is much higher than is possible in a conventional memory system. As we mentioned earlier, this database is only modestly sized for systems available in 1989. In the following sections, we show how multiprocessing can be used to achieve the performance necessary to display databases that are this size and larger.

18.4 INTRODUCTION TO MULTIPROCESSING

Displaying large databases at high frame rates clearly requires dramatic system performance, both in terms of computations and of memory of bandwidth. We have seen that the geometry portion of a graphics system can require more processing power than a single CPU can provide. Likewise, rasterization can require more bandwidth into memory than a single memory system can provide. The only way to attain such performance levels is to

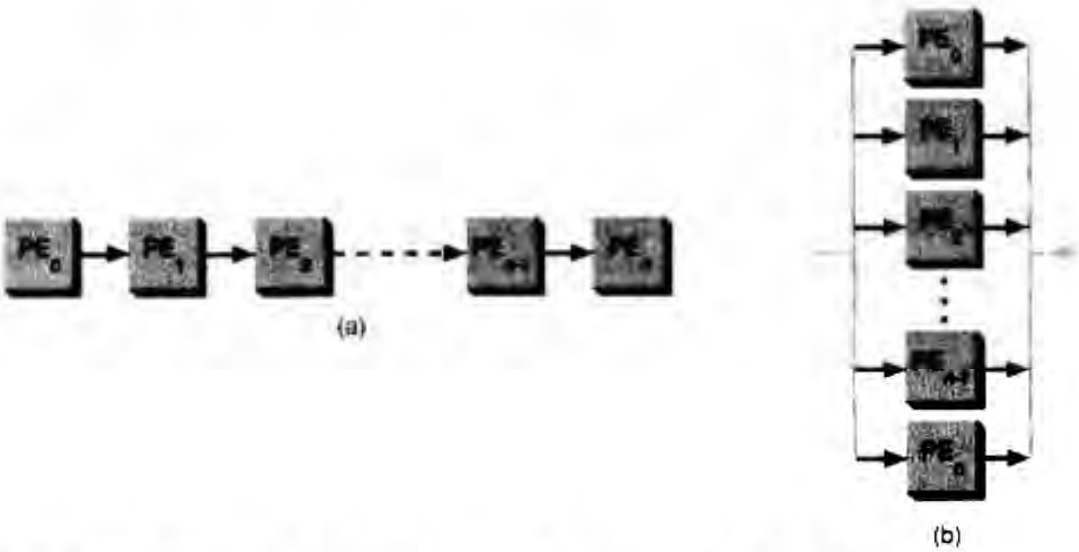


Fig. 18.9 Basic forms of multiprocessing: (a) pipelining, and (b) parallelism.

perform multiple operations concurrently and to perform multiple reads and writes to memory concurrently—we need concurrent processing.

Concurrent processing, or *multiprocessing*, is the basis of virtually all high-performance graphics architectures. Multiprocessing has two basic forms: *pipelining* and *parallelism* (we reserve the term *concurrency* for multiprocessing in general). A *pipeline processor* contains a number of *processing elements* (PEs) arranged such that the output of one becomes the input of the next, in pipeline fashion (Fig. 18.9a). The PEs of a *parallel processor* are arranged side by side and operate simultaneously on different portions of the data (Fig. 18.9b).

18.4.1 Pipelining

To pipeline a computation, we partition it into stages that can be executed sequentially in separate PEs. Obviously, a pipeline can run only as fast as its slowest stage, so the processing load should be distributed evenly over the PEs. If this is not possible, PEs can be sized according to the jobs they must perform.

An important issue in pipeline systems is *throughput* versus *latency*. Throughput is the overall rate at which data are processed; latency is the time required for a single data element to pass from the beginning to the end of the pipeline. Some calculations can be pipelined using a large number of stages to achieve very high throughput. Pipeline latency increases with pipeline length, however, and certain computations can tolerate only a limited amount of latency. For example, real-time graphics systems, such as flight simulators, must respond quickly to changes in flight controls. If more than one or two frames are in the rendering pipeline at once, the system's interactivity may be impaired, regardless of the frame rate.

18.4.2 Parallelism

To parallelize a computation, we partition the data into portions that can be processed independently by different PEs. Frequently, PEs can execute the same program. *Homogeneous* parallel processors contain PEs of the same type; *heterogeneous* parallel processors contain PEs of different types. In any parallel system, the overall computation speed is determined by the time required for the slowest PE to finish its task. It is important, therefore, to balance the processing load among the PEs.

A further distinction is useful for homogeneous parallel processors: whether the processors operate in *lock step* or independently. Processors that operate in lock step generally share a single code store and are called *single-instruction multiple-data* (SIMD) processors. Processors that operate independently must have a separate code store for each PE and are called *multiple-instruction multiple-data* (MIMD) processors.

SIMD processors. Because all the PEs in a SIMD processor share a single code store, SIMD processors are generally less expensive than MIMD processors. However, they do not perform well on algorithms that contain conditional branches or that access data using pointers or indirection. Since the path taken in a conditional branch depends on data specific to a PE, different PEs may follow different branch paths. Because all the PEs in a SIMD processor operate in lock step, they all must follow every possible branch path. To accommodate conditional branches, PEs generally contain an *enable register* to qualify write operations. Only PEs whose enable registers are set write the results of computations. By appropriately setting and clearing the enable register, PEs can execute conditional branches (see Fig. 18.10a).

Algorithms with few conditional branches execute efficiently on SIMD processors. Algorithms with many conditional branches can be extremely inefficient, however, since

```
statement 1;
if not condition then
  enable = FALSE;
statement 2;
toggle enable;
statement 3;
statement 4;
enable = TRUE;
statement 5;
statement 6;
```

Total operations:
10 if condition evaluates TRUE,
10 if condition evaluates FALSE

(a)

```
statement 1;
if condition then
  statement 2;
else
  begin
    statement 3;
    statement 4;
  end;
statement 5;
statement 6;
```

Total operations:
5 if condition evaluates TRUE,
6 if condition evaluates FALSE

(b)

Fig. 18.10 (a) SIMD and (b) MIMD expressions of the same algorithm. In a SIMD program, conditional branches transform into operations on the enable register. When the enable register of a particular PE is FALSE, the PE executes the current instruction, but does not write the result.

most PEs may be disabled at any given time. Data structures containing pointers (such as linked lists or trees) or indexed arrays cause similar problems. Since a pointer or array index may contain a different value at each PE, all possible values must be enumerated to ensure that each PE can make its required memory reference. For large arrays or pointers, this is an absurd waste of processing resources. A few SIMD processors provide separate address wires for each PE in order to avoid this problem, but this adds size and complexity to the system.

MIMD processors. MIMD processors are more expensive than SIMD processors, since each PE must have its own code store and controller. PEs in a MIMD processor often execute the same program. Unlike SIMD PEs, however, they are not constrained to operate in lock step. Because of this freedom, MIMD processors suffer no disadvantage when they encounter conditional branches; each PE makes an independent control-flow decision, skipping instructions that do not need to be executed (see Fig. 18.10b). As a result, MIMD processors achieve higher efficiency on general types of computations. However, since processors may start and end at different times and may process data at different rates, synchronization and load balancing are more difficult, frequently requiring FIFO buffers at the input or output of each PE.

18.4.3 Multiprocessor Graphics Systems

Pipeline and parallel processors are the basic building blocks of virtually all current high-performance graphics systems. Both techniques can be used to accelerate front-end and back-end subsystems of a graphics system, as shown in Fig. 18.11.

In the following sections, we examine each of these strategies. Sections 18.5 and 18.6 discuss pipeline and parallel front-end architectures. Sections 18.8 and 18.9 discuss pipeline and parallel back-end architectures. Section 18.10 discusses back-end architectures that use parallel techniques in combination.

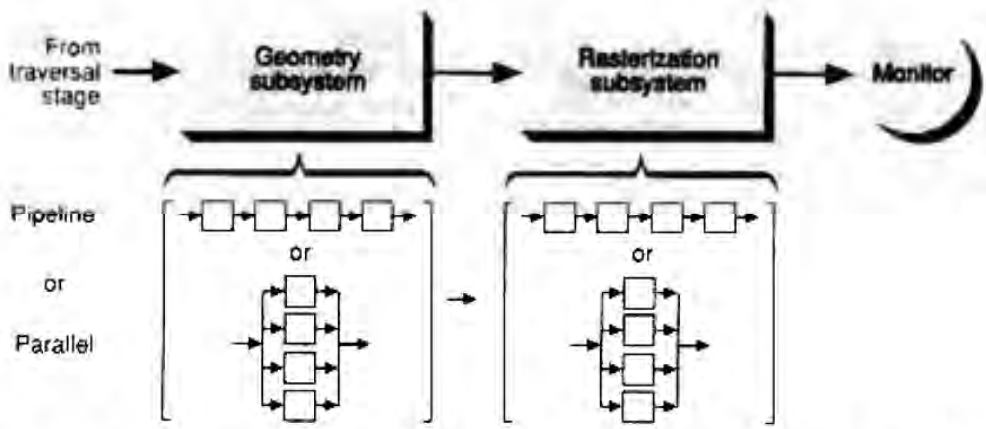


Fig. 18.11 Pipelining and parallelism can be used to accelerate both front-end and back-end portions of a graphics system.

18.5 PIPELINE FRONT-END ARCHITECTURES

Recall from Section 18.3 that the front end of a graphics display system has two major tasks: traversing the display model and transforming primitives into screen space. As we have seen, to achieve the rendering rates required in current applications, we must use concurrency to speed these computations. Both pipelining and parallelism have been used for decades to build front ends of high-performance graphics systems. Since the front end is intrinsically pipelined, its stages can be assigned to separate hardware units. Also, the large numbers of primitives in most graphics databases can be distributed over multiple processors and processed in parallel. In this section, we discuss pipeline front-end systems. We discuss parallel front-end systems in Section 18.6.

In introducing the standard graphics pipeline of Fig. 18.7, we mentioned that it provides a useful conceptual model of the rendering process. Because of its linear nature and fairly even allocation of processing effort, it also maps well onto a physical pipeline of processors. This has been a popular approach to building high-performance graphics systems since the 1960s, as described in [MYER68], a classic paper on the evolution of graphics architectures. Each stage of the pipeline can be implemented in several ways: as an individual general-purpose processor, as a custom hardware unit, or as a pipeline or parallel processor itself. We now discuss implementations for each stage in the front-end pipeline.

18.5.1 Application Program and Display Traversal

Some processor must execute the application program that drives the entire graphics system. In addition to feeding the graphics pipeline, this processor generally handles input devices, file I/O, and all interaction with the user. In systems using immediate-mode traversal, the display model is generally stored in the CPU's main memory. The CPU must therefore traverse the model as well as run the application. In systems using retained mode, the model is generally (but not always) stored in the display processor's memory, with the display processor performing traversal. Because such systems use two processors for these tasks, they are potentially faster, although they are less flexible and have other limitations, as discussed in Section 7.2.2.

Where very high performance is desired, a single processor may not be powerful enough to traverse the entire database with sufficient speed. The only remedy is to partition the database and to traverse it in parallel. This relatively new technique is discussed in Section 18.6.1.

18.5.2 Geometric Transformation

The geometric transformation stages (modeling transformation and viewing transformation) are highly compute-intensive. Fortunately, vector and matrix multiplications are simple calculations that require no branching or looping, and can readily be implemented in hardware.

The most common implementation of these stages is a single processor or functional unit that sequentially transforms a series of vertices. A pioneering processor of this type was the Matrix Multiplier [SUTH68], which could multiply a four-element vector by a

homogeneous transformation matrix in 20 microseconds. Other special-purpose geometry processors have been developed since then, most notably Clark's Geometry Engine, which can perform clipping as well (see Section 18.5.5). Recent geometry processors have exploited the power and programmability of commercial floating-point chips.

If pipelining does not provide enough performance, transformation computations can be parallelized in several ways:

- Individual components of a vertex may be calculated in parallel. Four parallel processors, each containing the current transformation matrix, can evaluate the expressions for x, y, z, and w in parallel.
- Multiple vertices can be transformed in parallel. If primitives are all of a uniform type—say, triangles—the three vertices of each triangle can be transformed simultaneously.
- Entire primitives can be transformed in parallel. If n transformation engines are available, each processor can transform every nth primitive. This technique has many of the advantages and disadvantages of parallel front-end systems, which we will discuss in Section 18.6.

18.5.3 Trivial Accept/Reject Classification

Trivial accept and reject tests are straightforward to implement, since they require at worst a dot product and at best a single floating-point comparison (or subtract) to determine on which side of each clipping plane each vertex lies. Because these tests require little computation, they are generally performed by the processor that transforms primitives.

18.5.4 Lighting

Like geometric transformation, lighting calculations are straightforward and are floating-point–intensive. A specialized hardware processor can calculate vertex colors based on a polygon's color and the light vector. More frequently, lighting calculations are performed using a programmable floating-point processor. In lower-performance systems, lighting calculations can be done in the same processor that transforms vertices. Note that if Phong shading is used, lighting calculations are deferred until the rasterization stage.

18.5.5 Clipping

Polygon clipping was once considered cumbersome, since the number of vertices can change during the clipping process and concave polygons can fragment into multiple polygons during clipping. Sutherland and Hodgman [SUTH74] showed that arbitrary convex or concave polygons can be clipped to a convex view volume by passing the polygon's vertices through a single processing unit multiple times. Each pass through the unit clips the polygon to a different plane. In 1980, Clark proposed unwrapping this processing loop into a simple pipeline of identical processors, each of which could be implemented in a single VLSI chip, which he named the *Geometry Engine* [CLAR82]. The Geometry Engine was general enough that it could transform primitives and perform perspective division as well.

Clipping using a Geometry Engine (or similar processor) can be performed either by a single processor that clips each polygon by as many planes as necessary, or by a pipeline of clipping processors, one for each clipping plane. The technique chosen affects the worst-case performance of the graphics system: Systems with only one clipping processor may bog down during frames in which large numbers of primitives need to be clipped, whereas systems with a clipping processor for each clipping plane can run at full speed. However, most of the clipping processors are idle for most databases and views in the latter approach.

General-purpose floating-point units recently have begun to replace custom VLSI transformation and clipping processors. For example, Silicon Graphics, which for many years employed custom front-end processors, in 1989 used the Weitek 3332 floating-point chip for transformations and clipping in their POWER IRIS system (described in detail in Section 18.8.2). The delicate balance between performance and cost now favors commodity processors. This balance may change again in the future if new graphics-specific functionality is needed and cannot be incorporated economically into general-purpose processors.

18.5.6 Division by *w* and Mapping to 3D Viewpoint

Like geometric transformation and lighting, the calculations in this stage are straightforward but require substantial floating-point resources. A floating-point divide is time consuming even for most floating-point processors (many processors use an iterative method to do division). Again, these stages can be implemented in custom functional units or in a commercial floating-point processor. In very high-performance systems, these calculations can be performed in separate, pipelined processors.

18.5.7 Limitations of Front-End Pipelines

Even though pipelining is the predominant technique for building high-performance front-end systems, it has several limitations that are worth considering. First, a different algorithm is needed for each stage of the front-end pipeline. Thus, either a variety of hard-wired functional units must be designed or, if programmable processors are used, different programs must be written and loaded into each processor. In either case, processor or functional-unit capabilities must be carefully matched to their tasks, or bottlenecks will occur.

Second, since the rendering algorithm is committed to hardware (or at least to firmware, since few systems allow users to reprogram pipeline processors), it is difficult to add new features. Even if users have programming support for the pipeline processors, the distribution of hardware resources in the system may not adequately support new features such as complex primitives or collision detection between primitives.

A final shortcoming of pipelined front ends is that the approach breaks down when display traversal can no longer be performed by a single processor, and this inevitably occurs at some performance level. For example, if we assume that traversal is performed by a 20-MHz processor and memory system, that the description of each triangle in the database requires 40 words of data (for vertex coordinates, normal vectors, colors, etc.), and that each word sent to the pipeline requires two memory/processor cycles (one to read it

from memory, another to load it into the pipeline), then a maximum of 20,000,000 / (2 · 40) = 250,000 triangles per second can be displayed by the system, no matter how powerful the processors in the pipeline are. Current systems are rapidly approaching such limits.

What else can be done, then, to achieve higher performance? The alternative to pipelining front-end calculations is to parallelize them. The following section describes this second way to build high-performance front-end systems.

18.6 PARALLEL FRONT-END ARCHITECTURES

Since graphics databases are regular, typically consisting of a large number of primitives that receive nearly identical processing, an alternate way to add concurrency is to partition the data into separate streams and to process them independently. For most stages of the front-end subsystem, such partitioning is readily done; for example, the geometric-transformation stages can use any of the parallel techniques described in Section 18.5.2. However, stages in which data streams diverge (display traversal) or converge (between the front end and back end) are problematic, since they must handle the full data bandwidth.

18.6.1 Display Traversal

Almost all application programs assume a single, contiguous display model or database. In a parallel front-end system, the simplest technique is to traverse the database in a single processor (serial traversal) and then to distribute primitives to the parallel processors. Unfortunately, this serial traversal can become the bottleneck in a parallel front-end system. Several techniques can be used to accelerate serial traversal:

- Traversal routines can be optimized or written in assembly code
- The database can be stored in faster memory (i.e., SRAM instead of DRAM)
- A faster traversal processor (or one optimized for the particular structure format) can be used.

If these optimizations are not enough, the only alternative is to traverse the database in parallel. The database either can be stored in a single memory system that allows parallel access by multiple processors (a shared-memory model), or can be distributed over multiple processors, each with its own memory system (a distributed-memory model).

The advantage of the shared-memory approach is that the database can remain in one place, although traversal must be divided among multiple processors. Presumably, each processor is assigned a certain portion of the database to traverse. Unfortunately, inherited attributes in a hierarchical database model mean that processors must contend for access to the same data. For example, each processor must have access to the current transformation matrix and to other viewing and lighting parameters. Since the data bandwidth to and from a shared-memory system may not be much higher than that of a conventional memory system, the shared-memory approach may not provide enough performance.

In the distributed-memory approach, each processor contains a portion of the database in its local memory. It traverses its portion of the database for each frame and may also perform other front-end computations. Distributing the database presents its own problems, however. Unless the system gives the application programmer the illusion of a contiguous database, it cannot support portable graphics libraries. Also, the load must be balanced

over the traversal processors if system resources are to be utilized fully. Hierarchical databases exacerbate both of these problems, since attributes in one level of a hierarchy affect primitives below them, and structures deep in a hierarchy may be referenced by multiple higher-level structure calls.

The following two sections examine two ways to distribute a hierarchical database over multiple processors: by structure, where each traversal processor is given a complete branch of the structure hierarchy; or by primitive, where each traversal processor is given a fraction of the primitives at each block in the hierarchy.

Distributing by structure. Distributing by structure is outwardly appealing, since state-changing elements in the structure apparently need to be stored only once. This can be an illusion, however, since multiple high-level structures may refer to the same lower-level substructure. For example, a database containing several cars, each described by a separate *car* structure, can be distributed by assigning each *car* structure to a separate processor. However, if each *car* structure refers to a number of *wheel* structures, *wheel* structures must also be replicated at every processor.

Load balancing among processors is also difficult. Since primitives in a structure are likely to be spatially coherent, changing the viewpoint or geometry within a scene may cause entire portions of the structure to be clipped or to reappear. Maintaining even loading among the multiple processors would require reassigning portions of the database dynamically.

Distributing by primitive. Distributing by primitive is costly, since the entire hierarchical structure of the database and any state-changing commands must be replicated at each processor. Structure editing is also expensive, since changes must be broadcast to every processor. Load balancing, however, is automatic. Since objects in a hierarchical database typically contain a large number of simple primitives (e.g., polygons forming a tiled surface), these primitives will be scattered over all the processors, and each processor will have a similar processing load.

Parallel display traversal is a relatively new technique. In 1989, the highest-performance architectures were just approaching the point where serial traversal becomes insufficient, and only a few systems had experimented with parallel traversal [FUCH89]. Neither of the distribution techniques for hierarchical databases that we have described is ideal. Compared to geometry processing, which easily partitions into parallel tasks, display traversal is much more difficult. Nevertheless, parallel traversal is likely to become increasingly important as system performance levels increase.

18.6.2 Recombining Parallel Streams

The transition between the front-end and back-end portions of the rendering pipeline is troublesome as well. In a parallel front-end system, the multiple streams of transformed and clipped primitives must be directed to the processor or processors doing rasterization. This can require sorting primitives based on spatial information if different processors are assigned to different screen regions.

A second difficulty in parallel front-end systems is that the ordering of data may change as those data pass through parallel processors. For example, one processor may transform two small primitives before another processor transforms a single, larger one. This does not

matter for many graphics primitives and rendering techniques. Certain global commands, however, such as commands to update one window instead of another or to switch between double buffers, require that data be synchronized before and after the command. If a large number of commands such as these occurs, some type of hardware support for synchronization may be necessary. A Raster Technologies system [TORB87] incorporates a special FIFO into each PE that stores tag codes for each command and allows commands to be resynchronized after they have been processed in separate PEs.

18.6.3 Pipelining versus Parallelism

We have seen that both pipelining and parallelism can be used to build high-performance front-end subsystems. Although pipelining has been the predominant technique in systems of the last decade, parallelism offers several advantages, including reconfigurability for different algorithms, since a single processor handles all front-end calculations, and more modularity, since PEs in a parallel system can be made homogeneous more easily than in a pipeline system. Because the performance of a pipeline system is limited by the throughput of its slowest stage, pipelines do not scale up as readily as do parallel systems. Parallel systems, on the other hand, require more complicated synchronization and load balancing and cannot use specialized processors as well as can pipelined systems. Both designs are likely to be useful in the future; indeed, the highest-performance systems are likely to combine the two.

18.7 MULTIPROCESSOR RASTERIZATION ARCHITECTURES

Recall that the output of the front-end subsystem is typically a set of primitives in screen coordinates. The rasterization (back-end) subsystem creates the final image by scan converting each of these primitives, determining which primitives are visible at each pixel, and shading the pixel accordingly. Section 18.2.4 identified two basic reasons why simple display-processor/frame-buffer systems are inadequate for high-performance rasterization subsystems:

1. A single display processor does not have enough processing power for all the pixel calculations.
2. Memory bandwidth into the frame buffer is insufficient to handle the pixel traffic—even if the display processor could compute pixels rapidly enough.

Much of the research in graphics architecture over the past decade has concerned ways to overcome these limitations. A great variety of techniques has been proposed, and many have been implemented in commercial and experimental systems. In this section, we consider low-cost, moderate-performance architectures that cast conventional algorithms into hardware. In Sections 18.8 and 18.9, we consider ways to improve performance by adding large amounts of parallelism to speed the calculation of the algorithm's "inner loop." In Section 18.10, we consider hybrid architectures that combine multiple techniques for improved efficiency or even higher performance. Figure 18.12 summarizes the concurrent approaches we shall discuss here.

Rasterization algorithm	*Architectural technique*		
	Serial pipeline	Highly parallel	Hybrid
Object order	Pipelined object order	Image parallel	Virtual buffer/ virtual processor
z-buffer, depth-sort, and BSP-tree algorithms	Polygon/edge/span-processor pipeline	Partitioned image memory Logic-enhanced memory	Parallel virtual buffer
Image order	Pipelined image order	Object parallel	
Scanline algorithms	Scan-line pipeline	Processor per primitive pipeline Tree-structured	Image composition

Fig. 18.12 Taxonomy of concurrent rasterization approaches.

18.7.1 Pipelined Object-Order Architectures

A direct way to add concurrency to rasterization calculations is to cast the various steps of a software algorithm into a hardware pipeline. This technique has been used to build a number of inexpensive, moderately high-performance systems. This approach can be used with either of the two main rasterization approaches: object order (z-buffer, depth-sort, and BSP-tree algorithms) and image order (scan-line algorithms). We consider object-order rasterization now and image-order rasterization in Section 18.7.2.

Object-order rasterization methods include the z-buffer, depth-sort, and BSP-tree algorithms (the z-buffer is by far the most common in 3D systems). The outer loop of these algorithms is an enumeration of primitives in the database, and the inner loop is an enumeration of pixels within each primitive. For polygon rendering, the heart of each of these algorithms is rasterizing a single polygon.

Figure 18.13 shows the most common rasterization algorithm for convex polygons. This algorithm is an extension of the 2D polygon scan-conversion algorithm presented in Section 3.6, using fixed-point arithmetic rather than integer arithmetic. Delta values are used to calculate the expressions for x, z, R, G, and B incrementally from scan line to scan line, and from pixel to pixel. We shall describe each step of the algorithm.

Polygon processing. Computations performed only once per polygon are grouped into this stage. The first step is to determine the initial scan line intersected by the polygon (this is determined by the vertex with the smallest y value). In most cases, the polygon intersects this scan line at a single pixel, with two edges projecting upward, the *left* and *right* edges. Delta values are calculated for x, z, R, G, and B for each edge. These delta values are sometimes called *slopes*.

Edge processing. Computations performed once for each scan line are grouped here. Scan lines within each primitive are processed one by one. The delta values computed previously are used to calculate x, z, R, G, and B values at the intersection points of the left and right edges with the current scan line (P_{left} and P_{right} in the figure). A contiguous sequence of pixels on a scan line, such as those between P_{left} and P_{right}, is called a *span*. Delta

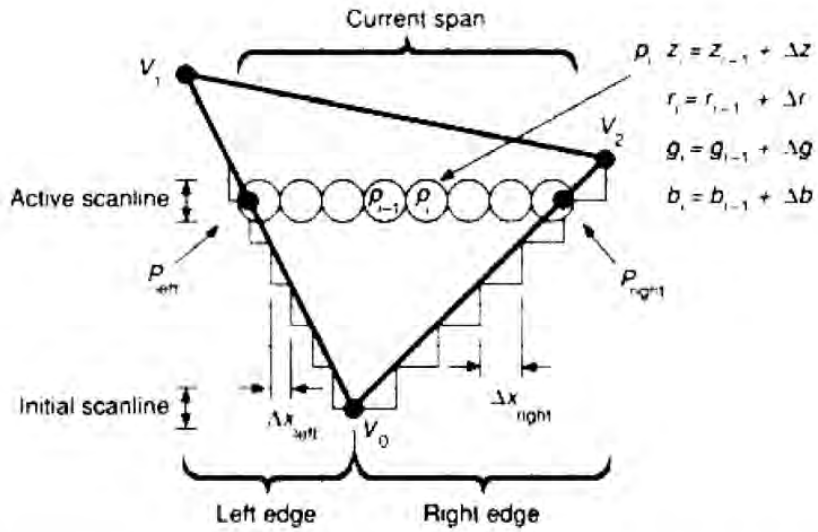


Fig. 18.13 Rasterizing a triangle. Each vertex (V_0, V_1, and V_2), span endpoint (P_{left} and P_{right}), and pixel (p_0, p_1, etc.) has z, R, G, and B components.

values for incrementing z, R, G, and B from pixel to pixel within the span are then calculated from the values at P_{left} and P_{right}.

Span processing. Operations that must be performed for each pixel within each span are performed here. For each pixel within the span, z, R, G, and B values are calculated by adding delta values to the values at the preceding pixel. The z value is compared with the previous z value stored at that location; if it is smaller, the new pixel value replaces the old one.

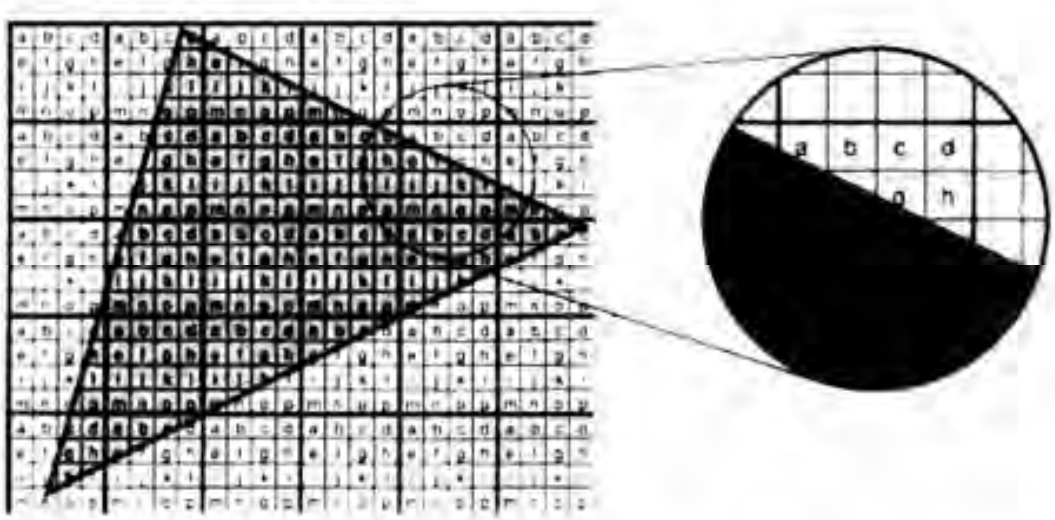


Fig. 18.14 A 4 × 4 interleaved memory organization. Each memory partition ("a" through "p") contains one pixel from each 4 × 4 block of pixels.

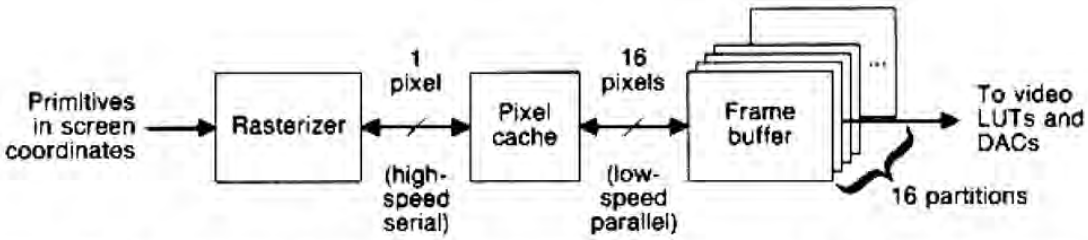


Fig. 18.15 A pixel cache matches bandwidth between a high-speed serial link to the rasterizer and a low-speed parallel link to the frame buffer.

A pipelined system containing one PE for each of the preceding three steps generates images dramatically faster than does a general-purpose display processor. In fact, it may generate pixels faster than a standard frame-buffer memory can handle. The Hewlett-Packard SRX [SWAN86], which uses this approach, generates up to 20 million pixels per second, approximately twice the speed of a typical VRAM memory system. In such systems, the rasterization bottleneck is access to frame-buffer memory.

Pixel cache. Pixels can be read and written faster if the frame buffer has some degree of parallel access. One way to accomplish this is to divide the memory into multiple—say, 16—partitions, each of which contains every fourth pixel of every fourth scan line, or perhaps every sixteenth pixel in every scan line (see Fig. 18.14). In this way, 16 pixels can be read or written in parallel. This technique, called *memory interleaving*, is also used in general-purpose CPU memory design.

A *pixel register* or *pixel cache* containing 16 pixels can be inserted between the rasterization pipeline and the interleaved image memory [GORI87; APGA88], as in Fig. 18.15. A cache allows the rasterizer to access individual pixels at high speed, assuming that the pixel is already in the cache. Multiple pixel values can be moved in parallel between the cache and the frame buffer at the slower speeds accommodated by the frame buffer.

As in any cache memory unit, performance depends on *locality of reference*, the principle that successive memory accesses are likely to occur in the same portion of memory. Erratic access patterns cause a high percentage of cache misses and degrade performance. For polygon rendering, the access pattern can be predicted precisely, since the extent of the polygon in screen space and the order in which pixels are generated are known before the pixels are accessed. Using this information, a cache controller can begin reading the next block of pixels from the frame buffer while the previous block of pixels is processed [APGA88].

Enhancing a rasterization subsystem with this kind of parallel-access path to frame-buffer memory may well increase the system throughput to the point where the bottleneck now becomes the single-pixel path between the rasterizer and the pixel cache. A logical next step is to enhance the rasterizer so that it can generate multiple pixel values in parallel. We consider such *image-parallel* architectures in Section 18.8.

18.7.2 Pipelined Image-Order Architectures

The alternative to object-order rasterization methods is *image-order* (or *scan-line*) rasterization, introduced in Section 15.4.4. Scan-line algorithms calculate the image pixel by pixel, rather than primitive by primitive. To avoid considering primitives that do not

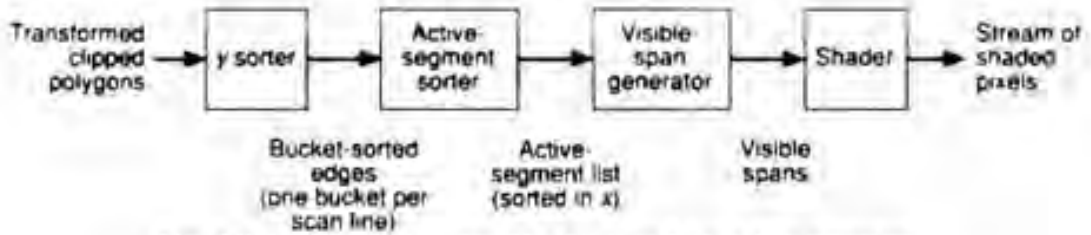


Fig. 18.16 Block diagram of a pipelined scan-line rasterizer.

contribute to the current scan line, most scan-line algorithms require primitives to be transformed into screen space and sorted into *buckets* according to the first scan line in which they each appear.

Scan-line algorithms can be implemented in hardware using the same approach as object-order algorithms: by casting the steps of the software algorithm into a pipelined series of hardware units. Much of the pioneering work on hardware scan-line systems was done at the University of Utah in the late 1960s [WYLI67; ROMN69; WATK70].

Figure 18.16 is a block diagram of a typical scan-line rasterizer. The *y sorter* places each edge of each polygon into the bucket corresponding to the scan line in which it first appears. The *active-segment generator* reads edges from these buckets, maintaining a table of active edges for the current scan line. From this table, it builds a list of active segments (a segment is a span within a single polygon), which is sorted by the x value of the left endpoint of each segment. The *visible-span generator* (called the *depth sorter* in the Utah system) traverses the active segment list, comparing z values where necessary, and outputs the sequence of visible spans on the current scan line. The *shader* performs Gouraud shading on these spans, producing a pixel stream that is displayed on the video screen.

Notice that no frame buffer is needed in this type of system, provided that the system can generate pixels at video rates. The original Utah scan-line system generated the video signal in real time for a modest number (approximately 1200) of polygons. However, since the rate at which pixels are generated depends on local scene complexity, a small amount of buffering—enough for one scan line, for example—averages the pixel rate within a single scan line. A double-buffered frame buffer allows complete independence of image-generation and image-display rates. This architecture was the basis of several generations of flight-simulator systems built by Evans & Sutherland Computer Corporation in the 1970s [SCHA83].

18.7.3 Limits of Pipeline Rasterization and the Need for Parallelism

Two factors limit the speedup possible in a pipeline approach. First, most rasterization algorithms break down easily into only a small number of sequential steps. Second, some of these steps are performed far more often than are others, particularly the steps in the inner loop of the rasterization algorithm. The processor assigned to these steps, therefore, becomes the bottleneck in the system.

The inner loop in an object-order (z-buffer) system is calculating pixels within spans; the inner loop in an image-order (scan-line) system is processing active edges on a scan line. For rasterization to be accelerated beyond the level possible by simple pipelining, these inner-loop calculations must be distributed over a number of processors. In z-buffer

systems, this produces image parallelism; in scan-line systems, this produces object parallelism. The following two sections discuss each of these approaches. Virtually all of today's high-performance graphics systems use some variation of them.

18.8 IMAGE-PARALLEL RASTERIZATION

Image parallelism has long been an attractive approach for high-speed rasterization architectures, since pixels can be generated in parallel in many ways. Two principal decisions in any such architecture are (1) how should the screen be partitioned? (into rows? into columns? in an interleaved pattern?), and (2) how many partitions are needed? In the following sections, we shall describe the most heavily investigated alternatives, discussing the advantages and disadvantages of each. Also, we shall identify which schemes are approaching fundamental limits in current architectures. Note that, because an image-parallel system rasterizes in object order, a frame buffer is required to store intermediate results.

18.8.1 Partitioned-Memory Architectures

Two obvious partitioning strategies are to divide pixels into contiguous blocks (Fig. 18.17a) [PARK80] and to divide them into an interleaved checkerboard pattern (Fig. 18.17b) [FUCH77a]. In either approach, a processor (or PE) is associated with each frame-buffer partition. Such organizations increase a graphics system's computation power by providing parallel processing, and its memory bandwidth by providing each PE with a separate channel into its portion of frame-buffer memory. During rasterization, polygons are transferred from the front end to the PEs in parallel, and each PE processes primitives in its portion of the frame buffer.

Contiguous partitioning. In the contiguous-region partitioning scheme, primitives need to be processed in only those regions in which they may be visible. These regions can be determined rapidly using geometric extents. If primitives are small compared to the region size, each primitive is likely to fall into a single region. Large primitives may fall into

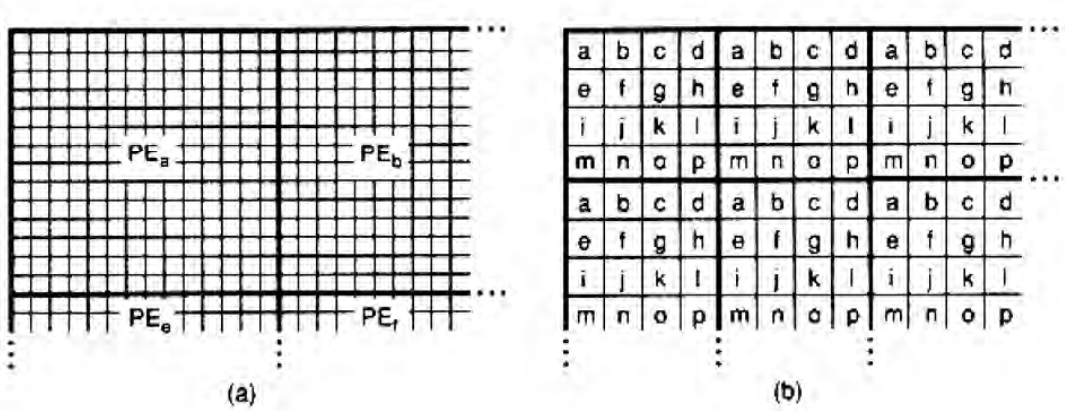


Fig. 18.17 Two schemes for frame-buffer partitioning. In (a), processors are assigned contiguous blocks of pixels; in (b), processors are assigned pixels in an interleaved pattern.

multiple regions. If the region size is chosen appropriately, the number of primitives handled by each processor will be approximately m/p, where m is the number of primitives in the database and p is the number of processors. Note, however, that if the viewpoint is chosen so unfortunately that all of the primitives fall into a single screen region, one processor must rasterize all of the primitives, and system performance decreases dramatically. In a contiguous region system, the frame rate is determined by the number of primitives in the busiest region.

Interleaved partitioning. Interleaved partitioning, on the other hand, achieves a better balance of workload, since all but the tiniest polygons lie in all partitions of the frame buffer. Since each processor handles every primitive (although only a fraction of its pixels), this scheme is less efficient in the best case than is the contiguous region approach. However, its worst-case performance is much improved, since it depends on the *total* number of primitives, rather than on the number in the busiest region. Because of this intrinsic load balancing, interleaved systems have become the dominant partitioned-memory architecture.

The polygon scan-conversion algorithm described in Section 18.7.1 requires set-up calculations to determine delta values and span endpoints before pixel computations can begin. These calculations need be performed only once per polygon or once per span, and can be shared among a number of PEs. The first proposed interleaved memory architectures [FUCH77a; FUCH79] contained no provision for factoring out these calculations from the PEs (see Fig. 18.18). Since each PE had to perform the entire rasterization algorithm for every polygon, many redundant calculations were performed.

Clark and Hannah [CLAR80] proposed an enhancement of this architecture to take advantage of calculations common to multiple PEs. In their approach, two additional levels of processors are added to perform polygon and edge processing. A single polygon processor receives raw transformed polygon data from the front-end subsystem and determines the polygon's initial scan line, slopes of edges, and so on. Eight edge processors (one per column in an 8×8 grid of pixel processors) calculate x, z, R, G, and B values at span endpoints. The edge processors send span information to the individual PEs (span processors), which interpolate pixel values along the span. The added levels of processing allow the PEs to perform only the calculations that are necessary for each pixel—a large improvement in efficiency. The rasterization portion of Silicon Graphics' recent high-performance systems uses this approach (see Section 18.8.2).

SIMD versus MIMD. A variation between systems of this type is whether PEs are SIMD or MIMD. Let us consider SIMD processors first. Figure 18.14 shows the mapping of processors to pixels in a 4×4 interleaved scheme. With a SIMD processor, the 16 PEs work on a contiguous 4×4 block of pixels at the same time. This arrangement is sometimes called a *footprint processor* because the 4×4 array of processors (the footprint) marches across the polygon, stamping out 16 pixels at a time. Notice that, if any pixel of a 4×4 block needs updating, the footprint must visit that block. For example, in the block of pixels shown in the inset of Fig. 18.14, processors a, b, c, d, g, and h must disable themselves, while processors e, f, i, j, k, l, m, n, o, and p process their respective pixels.

A disadvantage of SIMD processors is that they do not utilize their PEs fully. This occurs for two reasons. First, many of the PEs map to pixels outside the current primitive if

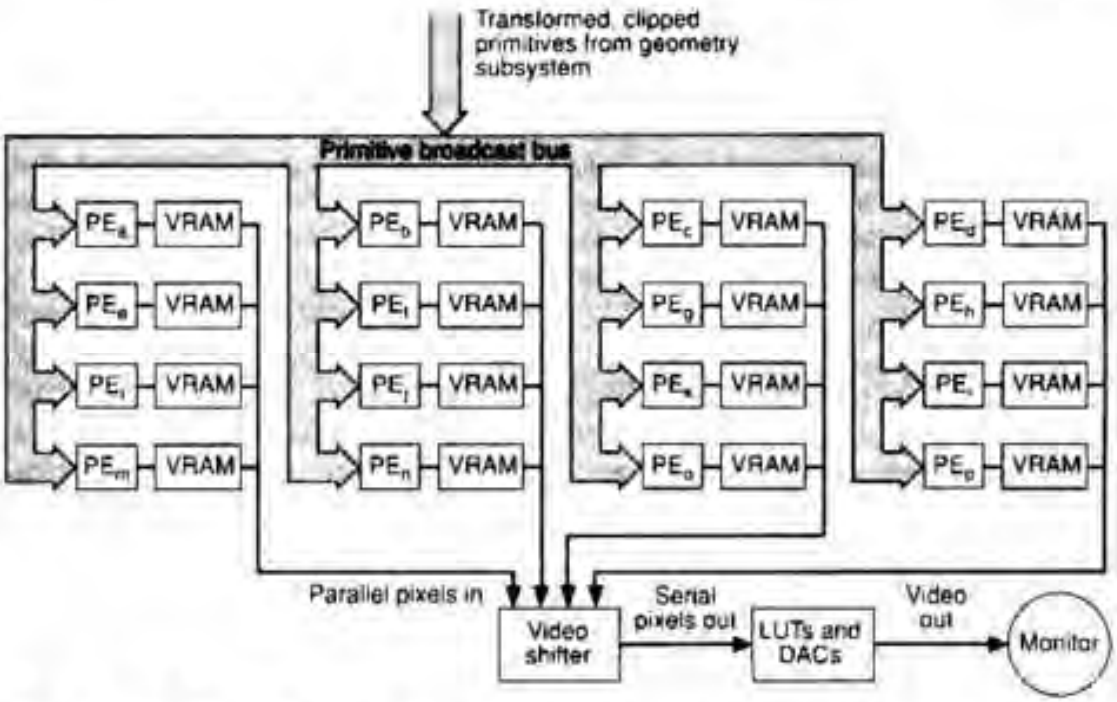


Fig. 18.18 Block diagram of a typical interleaved memory system. Each PE is responsible for one pixel in every 4 × 4 block of pixels.

small primitives are being rendered. For example, PEs in a 4 × 4 footprint processor rasterizing 100-pixel polygons map to pixels within the triangle as little as 45 percent of the time [APGA88]. Second, the choice of rasterization algorithm affects PE utilization. As remarked in Section 18.4.2, algorithms containing few conditional branches (including rasterizing convex polygons with Gouraud shading) can be implemented quite efficiently. Algorithms containing many conditional branches or using complicated data structures (such as rendering curved-surface primitives, texturing, shadowing, or antialiasing with a list of partially covering polygons) can be extremely difficult to make efficient. SIMD processors, however, can be built inexpensively and compactly, since a single code store and controller suffice for all the PEs. This offsets to some degree the poor PE utilization in a SIMD system. Several SIMD interleaved memory systems have been proposed and developed, including Gupta and Sproull's 8 by 8 Display [GUPT81b] and Stellar's GS2000 (see Section 18.11.3).

If we wish to support complex algorithms or to eliminate the idle processor cycles indicated in Fig. 18.14, we can add a control store to each PE, changing it from SIMD to MIMD. In a MIMD system, PEs do not need always to work on the same 4 × 4 pixel block at the same time. If each PE has FIFO input buffering, PEs can even work on different primitives. The separate control stores, FIFO queues, and hardware required to synchronize PEs add size and complexity to the system. Examples of successful MIMD interleaved-memory systems include AT&T's Pixel Machine [MCMI87] and Silicon Graphics' POWER IRIS (Section 18.8.2). Such systems compete well against SIMD systems on more complicated types of primitives or with more complicated rendering algorithms. For

example, AT&T's Pixel Machine Model PXM 924 ray traces simple scenes at interactive speeds—a feat unmatched by any SIMD system.

Since interleaved-memory architectures distribute the frame buffer over multiple processors, some provision must be made for scanning out pixels in the uninterrupted stream required by the video controller and display monitor. If the distributed frame buffer is constructed of VRAMs, this can be done in the manner shown in Fig. 18.18. Note that this is very similar to the technique described in Section 18.1.5 for implementing high-speed video scanout in a conventional frame-buffer memory.

Further subdivision. Suppose we wish to build a system with even higher performance. Since increasing the number of partitions in an interleaved-memory system increases the system's memory bandwidth and processing power, we might consider increasing the number of frame-buffer partitions—say, from 16 to 64, or even more. Unfortunately, the additional processors and datapaths required for each partition make such systems increasingly expensive.

An even more serious difficulty is supporting a larger number of partitions with the same number of frame-buffer memory chips. Each partition requires a minimum number of chips. For example, a partition with a 32-bit datapath between the PE and memory requires 8 4-bit wide chips, or 32 1-bit wide chips. Suppose we wish to build a 16-partition 1024- by 1024-pixel frame buffer with 128 bits per pixel. Using 256K × 4 VRAM chips, each partition requires 8 256K × 4 VRAM chips, so $16 \cdot 8 = 128$ chips are needed to support all 16 memory partitions. This is the exact number required to store the pixel data.

Suppose, however, that we increase the number of partitions from 16 to 64 (an 8 × 8 footprint). Although we still need only 128 memory chips to store the pixel data, we need $64 \cdot 8 = 512$ memory chips to support the PE–memory bandwidth. The extra 384 memory chips are needed only to provide communication bandwidth—not for memory. This is an extra expense that continues to grow as we subdivide the frame buffer further.

Increasing the density of memory parts from 1 Mbit to 4 Mbit exacerbates this problem even further. For example, if 1Mbit × 4 VRAM memory chips are used in the example mentioned above, 512 chips are still needed, even though each one contains sixteen times the memory actually required. Current systems such as the Silicon Graphics' POWER IRIS GTX (described in the next section), which uses 20 frame-buffer partitions, are already at the bandwidth limit. A way to ameliorate this problem would be for memory manufacturers to provide more data pins on high-density memory parts. Some 4-Mbit DRAM chips have eight data pins, rather than four, which helps somewhat, but only reduces the bandwidth problem by a factor of 2.

18.8.2 Silicon Graphics' POWER IRIS 4D/240GTX—An Interleaved Frame-Buffer Memory Architecture[2,3]

Silicon Graphics' POWER IRIS 4D/240GTX [AKEL88; AKEL89] uses many of the techniques described in this chapter. Like a number of its competitors, including the Ardent

[2]Material for this example is adapted from [AKEL88] and [AKEL89].

[3]In 1990, Silicon Graphics announced POWERVISION (similar to the GTX) that renders 1 million Gouraud-shaded triangles/sec and with 268 bits/pixel for antialiasing and texturing.

Titan [BORD89], the Megatek Sigma 70 [MEGA89], and the Stellar GS2000 (Section 18.11.3), the SGI POWER IRIS is a high-end graphics workstation, designed to combine general-purpose processing and high-speed 3D graphics for engineering and scientific applications.

The POWER IRIS has a powerful general-purpose CPU composed of four tightly coupled multiprocessors sharing a single memory bus. Its graphics subsystem can render over 100,000 full-colored, Gouraud-shaded, z-buffered quadrilaterals per second [AKEL89]. The POWER IRIS continues Silicon Graphics' tradition of immediate-mode display traversal, aggressive use of custom VLSI, hardware front-end pipeline, and interleaved-memory frame-buffer architecture. The POWER IRIS's architecture, diagrammed in Fig. 18.19, is composed of five major subsystems:

1. *CPU subsystem*—runs the application and traverses the display model
2. *Geometry subsystem*—transforms and clips graphical data to screen coordinates
3. *Scan-conversion subsystem*—breaks points, lines, and polygons into pixels
4. *Raster subsystem*—computes visibility and writes pixel data to frame buffer
5. *Display subsystem*—displays contents of frame buffer on color monitor.

CPU subsystem. The *CPU subsystem* runs the application and traverses the database. It is composed of four tightly coupled, symmetric, shared-memory multiprocessors. Hardware provides high-speed synchronization between processors, so parallelism can be achieved within a single process (although special programming constructs are required).

Geometry subsystem. The *geometry subsystem* transforms, clips, and lights primitives. It is composed of five floating-point processors arranged in a pipeline. Each of these processors, called a *geometry engine* (GE), contains an input FIFO, a controller, and a floating-point unit capable of 20 MFLOPS. Unlike Silicon Graphics' earlier Geometry Engine (see Section 18.5.4), the POWER IRIS' GEs are based on a commercial floating-point chip, the Weitek 3332.

The first GE transforms vertices and vertex normals. The second GE performs lighting calculations (supporting up to eight point light sources). The third GE performs trivial accept/reject clipping tests. The fourth GE performs exact clipping on primitives that cross clipping boundaries, and also does perspective division for all primitives. The fifth GE clips color components to maximum representable values, calculates depth-cued colors where necessary, and converts all coordinates to screen-space integers.

Scan-conversion subsystem. The *scan-conversion subsystem* rasterizes primitives using the pipeline approach described in Section 18.7.1, except that its spans are vertical columns of pixels, rather than the horizontal rows we have assumed so far (the only effect on the rasterization algorithm is that x and y coordinates are interchanged).

The single *polygon processor* sorts the vertices of each polygon from left to right in screen space. The sorted vertices are then used to decompose the polygon into vertically aligned trapezoids. The upper pair of vertices and the bottom pair of vertices of each trapezoid are used to calculate slopes for use by the edge processors.

The *edge processor* uses vertex and slope information to compute x, y, and z coordinates and color values for each pixel that lies on the top or bottom edges of each

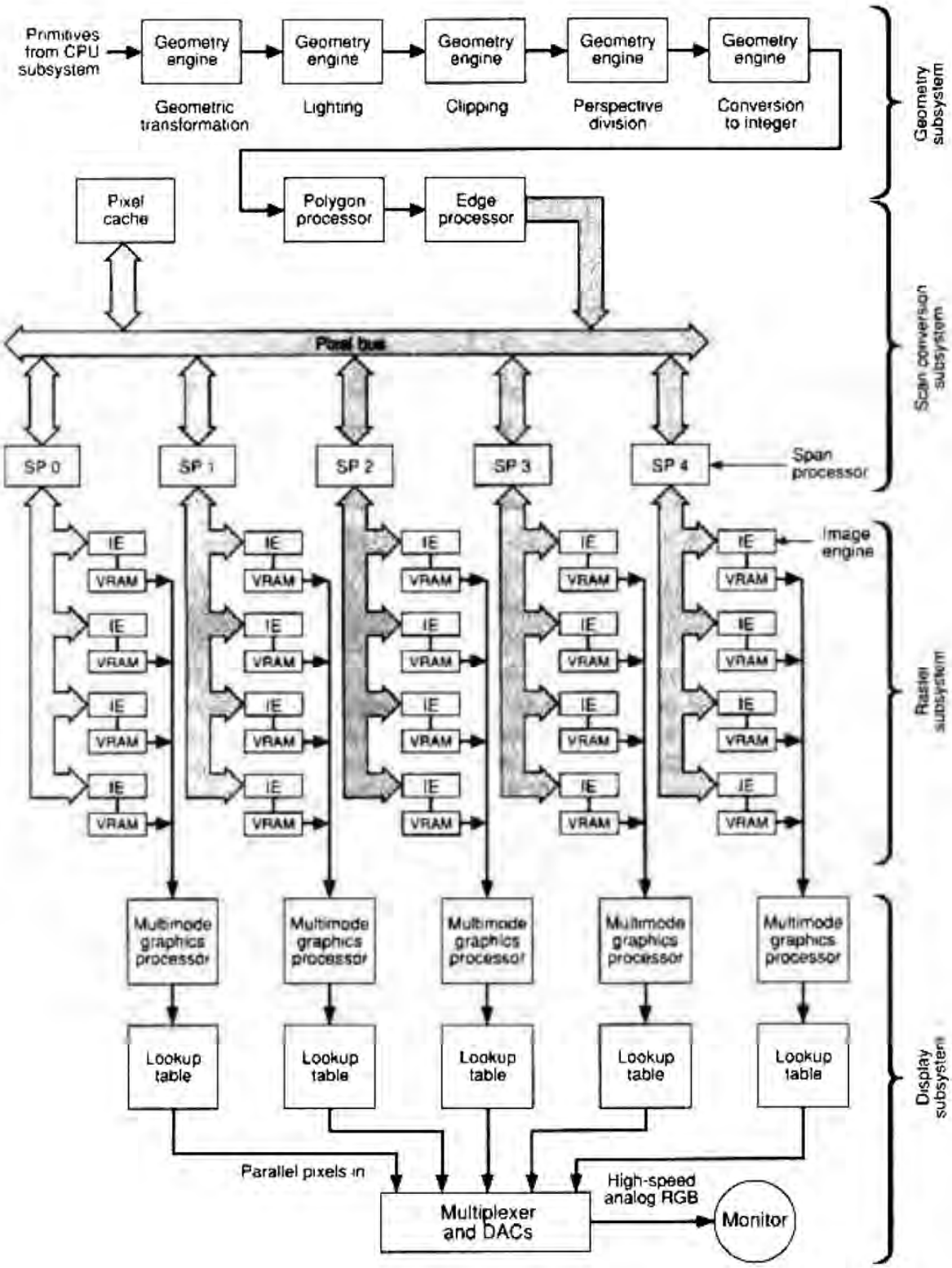


Fig. 18.19 Silicon Graphics' GTX system architecture (based on [AKEL88]).

trapezoid. In a given vertical column, a pair of pixels determines the top and bottom endpoints of a vertical span. These pixel pairs, together with slope information, are passed on to the span processors.

Each of the five parallel *span processors* is responsible for one-fifth of the columns of the display screen. For example, span processor 0 manages scan lines 0, 5, 10, and so on. Span processors calculate z, R, G, B, and α (for transparency and antialiasing) values for each pixel in the span. Because spans generated from a single polygon are adjacent, the processing load over span processors is approximately uniform.

The *pixel cache* buffers blocks of pixels during pixel copy operations so that the full bandwidth of the pixel bus can be used.

Raster subsystem. The *raster subsystem* takes pixel data generated by the span processors and selectively updates the image and z bitplanes of the frame buffer, using the results of a z comparison and α blending. The raster subsystem is composed of 20 *image engines*, each responsible for one-twentieth of the screen's pixels, arranged in a 4- × 5-pixel interleaved fashion. 96 bits are associated with each pixel on the screen: two 32-bit image buffers (r, G, B, and α), a 24-bit z-buffer, four overlay/underlay bitplanes, and four window bitplanes.

The overlay/underlay bitplanes support applications that use pop-up menus or windowing backgrounds. The window bitplanes define the display mode (single- or double-buffered, etc.) and window-masking information.

The 20 image engines work on pixels in parallel. Each can blend values based on the pixel's α value, allowing transparent or semitransparent objects to be displayed, and allowing supersampling antialiasing.

Display subsystem. The display subsystem contains five *multimode graphics processors* (MGPs), each assigned one-fifth of the columns in the display. The MGPs concurrently read image data from the frame buffer (together with window-display-mode bits), process them using the appropriate display mode (RGB or pseudocolor), and send them on to digital-to-analog converters for display.

The GTX's architecture is a good example of many of the techniques we have discussed so far. It provides high performance for polygon rendering at a reasonable cost in hardware. Because the GTX's rendering pipeline is highly specialized for graphics tasks, however, the system has difficulty with the advanced rendering techniques we shall discuss in Section 18.11, and its resources cannot be applied easily to nongraphics tasks. Section 18.11.3 discusses the architecture of Stellar's GS2000, which has complementary advantages and disadvantages.

18.8.3 Logic-Enhanced Memory

Since commercial memories may not support enough frame-buffer partitions, one might consider building custom memories with a large number of concurrently accessible partitions on a single chip. Since each (intrachip) partition must have its own connection to its associated (external) processor, extra pins must be added to each memory package to support these additional I/O requirements. Alternatively, multiple processors could be built

onto the chip itself. The first possibility—that of adding pins to memory chips—directly increases the memory bandwidth, but makes the chip package and associated circuit boards larger and more expensive, and also increases the power requirements. These packaging effects become progressively more severe as memory densities increase. (In the past two decades, the number of bits in a typical RAM has increased by a factor of 1000, while the size of the package and the number of pins have changed hardly at all.)

In this section, we shall concentrate on the second option—that of adding processing to multipartition memory chips. In the simplest schemes, only new addressing modes are provided, such as the ability to address an entire rectangle of memory pixels in parallel. At the other extreme, an entire microprocessor (including code store) could be provided for each internal partition of memory.

Before we describe specific logic-enhanced–memory approaches, let us consider the advantages and disadvantages of any logic-enhanced–memory scheme. First, adding logic or processing to memories has the potential to increase vastly the processing power within a system. By increasing the number of internal memory partitions and providing processing for each on the same chip, enormous processor/memory bandwidths can be achieved. Second, in custom VLSI chips, options become available that are impractical in board-level systems, since VLSI technology has an entirely different set of cost constraints for gates, wiring channels, and memory. Third, off-chip I/O bandwidth can potentially be reduced, since the only off-chip communication needed is to control the processor and to scan pixels out of the chip; this translates into fewer pins in the package and thus to a smaller package and less board space.

The principal disadvantages of an enhanced-memory approach are low memory densities and increased cost. With enormous production volumes, commercial DRAM manufacturers can afford to develop specialized, high-density fabrication capabilities and to incur large development costs to fine-tune their designs. Design and fabrication resources for custom memory chips, however, are generally more limited, resulting in densities lower than those of commercial RAMs. The price per chip is also high, since the costs for designing a custom VLSI chip are not offset by such large sales volumes. In spite of these disadvantages, at least one custom memory chip for graphics has become commercially successful—the VRAM. It remains to be seen whether other custom memory designs for graphics have sufficient market appeal to justify large-scale commercial development.

Pixel-Planes. An early and very general logic-enhanced–memory design is Pixel-Planes [FUCH81]. Pixel-Planes pushes frame-buffer subdivision to its extreme: It provides a separate processor for every pixel in the display. Each SIMD *pixel processor* is a 1-bit processor (ALU) with a small amount of memory. Figure 18.20 shows a block diagram of an enhanced-memory chip in Pixel-Planes 4, a prototype system completed in 1986 [EYLE88]. Its design is similar to that of the VRAM chip of Fig. 18.3, only here the 1-bit ALUs and associated circuitry replace the video shifter. Each enhanced-memory chip contains 128 pixels (columns in the memory array), and each pixel contains 72 bits of local memory (rows within the column).

Pixel-Planes' performance is not based simply on massive parallelism. If it was, each PE would have to perform all the operations for scan conversion independently, resulting in many redundant calculations and a grossly inefficient system. Rather, Pixel-Planes uses a

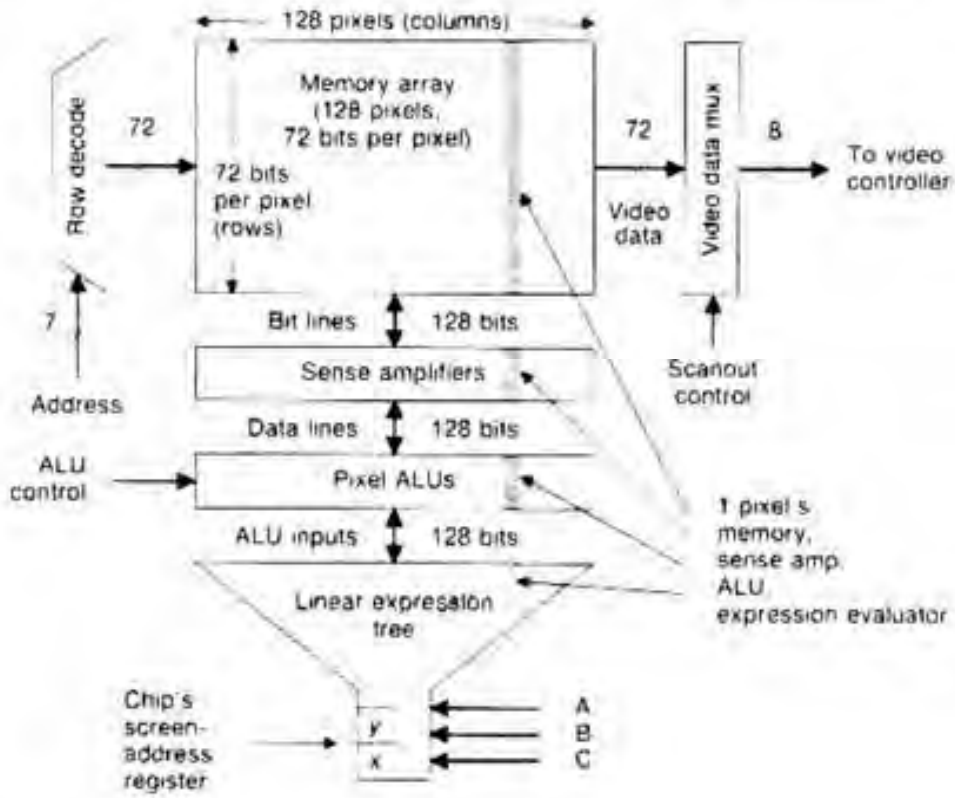


Fig. 18.20 Pixel-Planes 4 logic-enhanced–memory chip.

global computing structure called a *linear expression tree* that evaluates linear expressions of the form $F(x, y) = Ax + By + C$ for every pixel (x, y) of the screen in parallel [FUCH85]. A, B, and C floating-point coefficients are input to the tree; each pixel receives its own value of F in its local memory, 1 bit per clock cycle (approximately 20–30 cycles are required for each linear expression). The linear expression tree is especially effective for accelerating rasterization calculations, since many of these can be cast as linear expressions. For example,

- Each edge of a convex polygon can be described by a linear expression. All points (x, y) on one side of the edge have $F(x, y) \geq 0$; all points on the other side of the edge have $F(x, y) \leq 0$.
- The z value of all points (x, y) within a triangle can be described as a linear expression.
- R, G, and B color components of pixels in a Gouraud-shaded triangle can be described as linear expressions.

Double-buffering is implemented within Pixel-Planes chips by providing a *video-data multiplexer* that reads pixel values from specific bits of pixel memory while the image is computed in the remaining bits. Video data are scanned out of the chip on eight video data pins.

Displaying images on Pixel-Planes requires modifying the algorithms we have assumed so far. In addition to transforming and clipping primitives in the usual manner, the

front-end subsystem must compute coefficient sets for the linear equations describing each primitive's edges, z values, and color values. Also, rasterization proceeds in parallel, since large areas of pixels can be affected at once using the linear expression tree. The following section describes a sample algorithm on Pixel-Planes.

Rasterizing a triangle on Pixel-Planes. Here, we briefly describe the algorithm to display Gouraud-shaded triangles on a Pixel-Planes system. Pixel-Planes can display more general polygons, although the algorithms are somewhat more complicated.

Scan conversion. Figure 18.21 shows the steps in scan converting a triangle. The first step is to enable all the pixel processors in the display. Edges are encoded as linear expressions $F(x, y) = Ax + By + C = 0$, as described previously. Each expression is then evaluated in parallel at every pixel in the screen, using the linear expression tree. Each pixel processor tests the sign of F to determine whether it lies on the proper side of the edge. If it lies outside the edge, the pixel processor disables itself by setting its enable bit to 0. After all the edges of a polygon have been tested, the only pixel processors still enabled are those lying within the polygon. These pixel processors alone participate in visibility and shading calculations.

z-buffering. After a polygon has been scan converted, Pixel-Planes evaluates the linear expression for z for all pixels in parallel. Each pixel processor compares this new z value with the one stored in its z-buffer. If the new z value is smaller, the current polygon is visible at the pixel; the pixel processor updates its z-buffer and remains enabled. If the new z value is larger, the pixel disables itself and does not participate in shading calculations.

Gouraud shading. The linear expressions for R, G, and B components of the color are evaluated for each pixel in parallel by the linear expression tree. Pixel processors that are still enabled write the new color components into their pixels' color buffers.

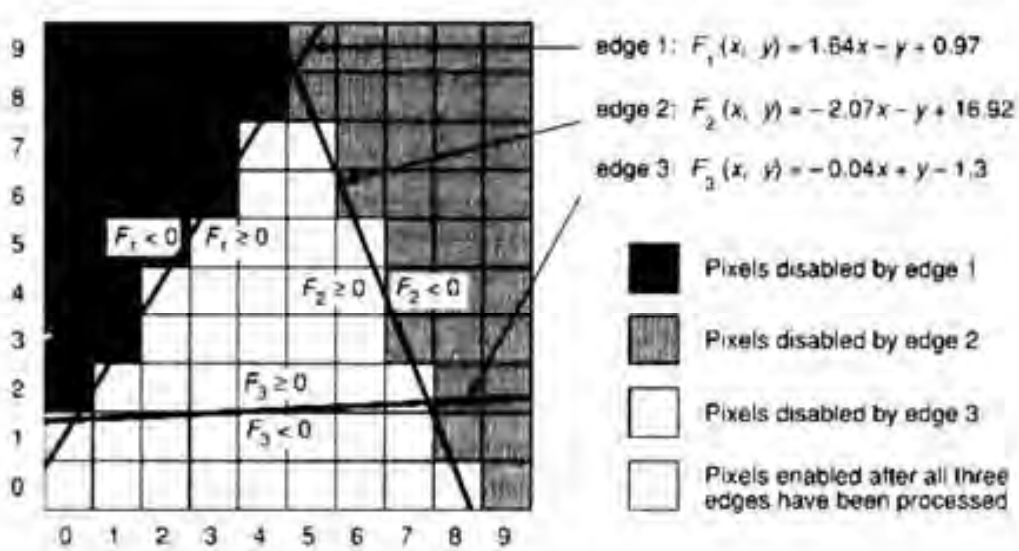


Fig. 18.21 Rasterizing a triangle on Pixel-Planes.

Note that scan conversion in Pixel-Planes is completely independent of a polygon's size, so a large polygon is rasterized as rapidly as a small one. Note, also, that operations that cannot be expressed as linear equations may take much longer to execute on Pixel-Planes than those that can (e.g., a quadratic expression can be calculated in pixel memory by multiplying values 1 bit at a time). Nevertheless, efficient algorithms for drawing spheres, casting shadows, antialiasing, and texture mapping have been developed for Pixel-Planes [FUCH85].

A full-scale system, Pixel-Planes 4, containing 262,144 processors forming a 512- by 512-pixel image, was completed in 1986. Although its performance was impressive for its time (40,000 Gouraud-shaded triangles of arbitrary size per second), it contained 2048 custom memory chips—a prohibitive expense for a commercial graphics system. This highlights the fundamental disadvantage of such a highly parallel SIMD approach: the very low utilization of pixel processors. Since all the PEs work in lock step, they cannot be retargeted to other tasks, even if only a few of them are still calculating useful results. This problem is especially severe when large numbers of small polygons are being drawn, because the first steps of the scan-conversion process disable almost all the screen's pixel processors.

Several logic-enhanced–memory graphics architectures have been developed that are more frugal in their use of silicon, at some sacrifice in either speed or generality. Although neither of the architectures described next directly supports the 3D rendering techniques assumed to this point, both provide very high performance in their respective domains (displaying 2D rectangles and generating 2D halftone images), and both provide insight into the potential of the logic-enhanced–memory approach.

Rectangle area-filling memory chip. Whelan proposed modifying the row and column addressing in the 2D memory-cell grid of a typical RAM to allow an entire rectangular region to be addressed at once [WHEL82]. Minimum and maximum row and column addresses specify the left, right, top, and bottom boundaries of the region. One write operation can store a single data value in every location within the region. This allows upright, constant-shaded rectangles to be rasterized in very few clock cycles—just enough to specify the four address values and the constant data.

Scan Line Access Memory. Demetrescu designed a more complicated chip called a Scan Line Access Memory (SLAM) for rasterizing more general 2D primitives [DEME85]. Like VRAMs and Pixel-Planes, SLAM takes advantage of the fact that, internally, a RAM reads or writes an entire row of its memory array in one cycle. Figure 18.22 shows a block diagram of a single SLAM chip. Each chip contains 256 × 64 bits of frame-buffer memory. Each row of memory corresponds to 1 bit in a scan line of pixels. In a system with k bits per pixel, a SLAM chip can store up to $64/k$ scan lines of 256 pixels each. In each memory cycle, a SLAM chip can read or write one row of memory from its memory array. By specifying appropriate x_{min} and x_{max} values, one can address any contiguous span of pixels on the current scan line, allowing fast polygon scan conversion. Video scanout is accomplished using a display shifter in exactly the same manner as a VRAM chip.

In a single clock cycle, either a row address, an x_{min} value, an x_{max} value, or a 16-bit repeating data pattern (for specifying halftone patterns) can be specified. Concurrent with

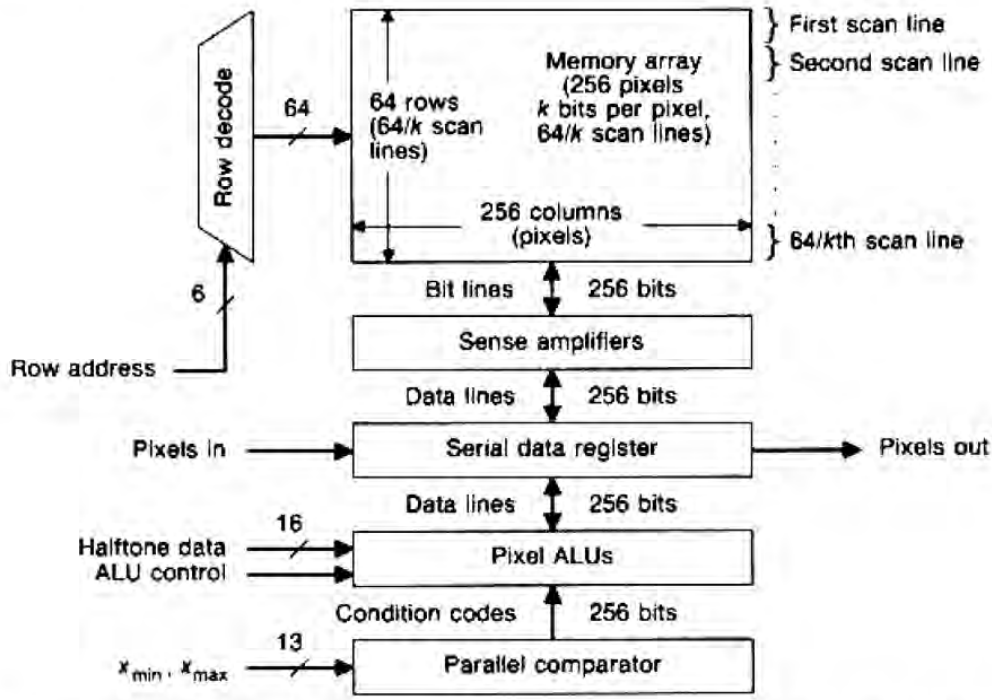


Fig. 18.22 Block diagram of a SLAM chip (for a system configured with k bits per pixel).

any one of these commands, SLAM can write the current scan-line segment into its memory array and can optionally increment the row address. Since, for many primitives, row addresses and halftone patterns need to be specified only once (for the initial scan line), succeeding scan lines can be processed rapidly. SLAM therefore can scan convert a convex polygon covering n scan lines in $2n + 2$ cycles (four cycles to specify the row address, x_{min}, x_{max}, and the halftone pattern for the initial scan line, and two cycles to specify x_{min} and x_{max} for each of the remaining $n - 1$ scan lines).

A SLAM system is composed of a number of SLAM chips (the number depends on the dimensions of the display screen). Figure 18.23 shows a SLAM system for updating a 512 by 512 monochrome display screen. Systems with more bits per pixel require proportionately more SLAM chips. SLAM can also be extended to display Gouraud-shaded polygons by adding a Pixel-Planes–style linear-expression tree. However, this enhanced version of SLAM would require approximately the same amount of hardware as Pixel-Planes and would suffer the same low utilization of PEs, since the pixels in any given (small) primitive would likely be contained in just a few SLAM chips.

Although both Whelan's architecture and the original SLAM architecture use less hardware than does Pixel-Planes, they do not offer the generality necessary to render realistic 3D images. Pixel-Planes and the enhanced version of SLAM do offer this generality, but suffer poor PE utilization. It would be useful to gain the performance of these processor-per-pixel architectures, but with higher PE utilization for small primitives. Section 18.10.1 examines a way to accomplish this using enhanced-memory arrays smaller than the full screen size.

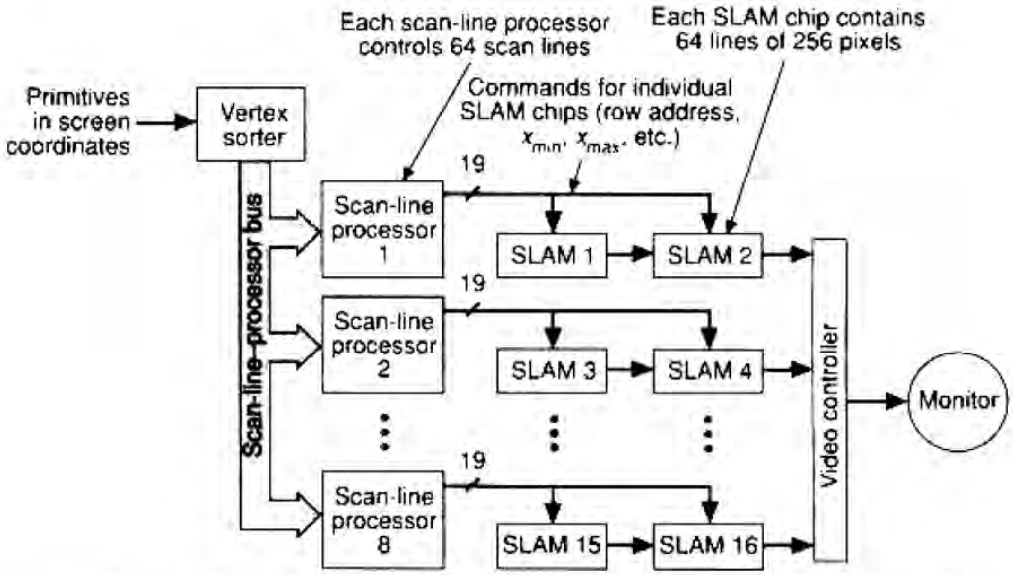


Fig. 18.23 SLAM system for updating a 512- by 512-pixel monochrome display.

18.9 OBJECT-PARALLEL RASTERIZATION

So far, we have focused on image-parallel architectures. The *object-parallel* family of parallel architectures parallelizes the inner loop of image-order (generally scan-line) algorithms. In an object-parallel architecture, multiple primitives (objects) are processed in parallel, so that final pixels may be generated more rapidly.

The usual object-parallel approach is to assign primitives (either statically or dynamically) to a number of homogeneous *object processors*, each of which can generate an entire image containing its primitive(s). During rasterization, each object processor enumerates the pixels of the display in some specific order (generally scan-line), generating color, z, and possibly partial-coverage values for its primitive(s). The pixel streams from each of the object processors are then combined to produce a single pixel stream for the final image. Although any number of primitives may be assigned to each object processor, most designs allocate a single primitive per processor. The advantage is that each processor can perform a well-defined task and thus can be reproduced inexpensively.

General Electric's NASA II. A pioneering real-time processor-per-primitive system was General Electric's NASA II flight simulator [BUNK89], delivered to NASA in 1967. The NASA II contained a number of hardware units called *face cards*, each of which rasterized a single polygon at video rates. At any given instant, each face card would process the same pixel.

The NASA II used a depth-sort visibility algorithm, rather than a z-buffer. The output of each face card included a bit indicating whether or not the pixel was covered by its polygon, the pixel color, and the polygon priority number. This information was fed into a priority multiplexer so that, at each pixel, the color of the highest-priority visible polygon was output. Since face cards were expensive, they were reassigned to new polygons when their polygons no longer intersected the current scan line. The NASA II system

could display a maximum of 60 polygons on any given scan line.

Later processor-per-primitive designs have adopted z-comparison visibility algorithms that allow greater flexibility. Processors are typically arranged in a pipeline or binary tree, so that streams of pixels can be merged to produce the pixel stream of the final image.

18.9.1 Processor-per-Primitive Pipelines

A simple way to combine the multiple pixel color and z streams produced by the multiple object processors is to pipeline the processors so that a stream of color values and z values passes between them (Fig. 18.24). Each processor generates color and z values for its primitive at the current pixel, and compares its z value with the incoming z value. If the incoming z value is smaller, the color and z values are passed on unchanged; if it is greater, the primitive's color and z values are sent out. The output of the last processor in the pipeline is the video stream of the final image.

Cohen and Demetrescu [DEME80] designed what was perhaps the first processor-per-polygon pipeline system. In this design, each polygon is assigned a polygon processor for the duration of the frame-generation time. Weinberg [WEIN81] proposed an enhancement of this architecture to generate antialiased images. Instead of passing pixel color and z values between adjacent processors, Weinberg's design passes, for each pixel, an arbitrarily long packet of polygon-fragment information relating to that pixel. Each polygon processor compares z and edge information for its polygon with the incoming packet describing the pixel. The packet is updated to take the new polygon into account, and is forwarded to the next polygon processor. A set of filtering engines at the end of the pipeline calculates the single color value for each pixel from the (final) packet associated with that pixel.

A team at Schlumberger proposed a design combining aspects of the NASA II, Cohen–Demetrescu, and Weinberg designs [DEER88]. Their Triangle Processor and Normal Vector Shader uses a pipeline of triangle processors that passes along surface-normal and polygon-color data, rather than actual pixel colors. Triangles are assigned to processors only during scan lines in which they are active in the same manner as in the NASA II. A pipeline of Normal Vector Shaders computes the Phong lighting model for pixels emerging from the triangle-processor pipeline. Because lighting and shading calculations are delayed until the end of the pipeline, Phong calculations need to be performed only once for each pixel in the final image, rather than once for each pixel of every polygon—a substantial savings.

18.9.2 Tree-Structured Processor-per-Primitive Architectures

An alternative to pipelining object processors is to arrange them in parallel and to use a binary tree of compositors to merge the color and z streams. This approach allows

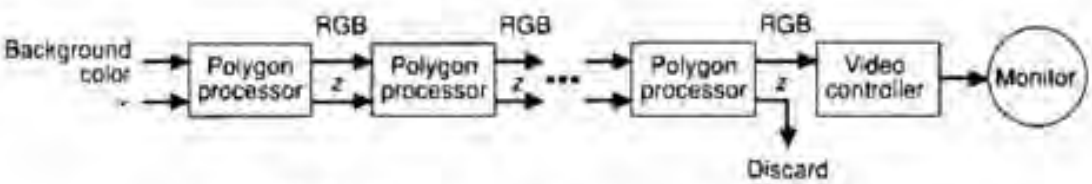


Fig. 18.24 Processor-per-polygon pipeline system.

rasterization within the object processors to be truly synchronous (there is no pipeline delay between the times at which two processors compute the same pixel), and reduces latency incurred by a long pipeline. However, special logic is required to perform the composition operation.

Fussell [FUSS82] proposed the first binary-tree-structured processor-per-primitive design. He observed that, since z comparators are simple, they can be made to run at video rates. Shaw, Green, and Schaeffer [SHAW91] proposed a way to antialias images generated using Fussell's scheme. In their approach, custom VLSI *compositors* combine images at each of the nodes in the image-composition tree. These compositors implement a simplified version of Duff's image-composition algorithm (see Section 17.6), providing proper treatment of many partial-pixel-coverage cases. A disadvantage of this approach is that the antialiasing is not perfect; color bleedthrough between abutting polygons is possible, whereas this does not occur if z values alone are used to determine visibility.

Kedem and Ellis's Ray Casting Machine [KEDE84] directly rasterizes images from a constructive solid geometry (CSG) tree representation (see Chapter 7). The Ray Casting Machine maps a CSG tree to hardware by assigning each CSG primitive to a processor called a *primitive classifier* and each operator to a processor called a *classification combiner*.

The image is traversed pixel by pixel in raster-scan order. For each pixel, primitive classifiers compute the segment of the ray through that pixel that is interior to their respective CSG primitives. These segments are then passed upward to the classification combiners, which perform set operations on them using the techniques described in Section 15.10.3. Segments are split and combined as needed. The updated segments are passed upward in the tree until they reach the root of the tree.

The set of segments emerging from the root of the classification-combiner tree describes the intersection of the current ray with the entire CSG object. The near endpoint of the nearest segment contains the z value of the visible surface at that pixel. This value is used to compute a color value for the pixel using any desired lighting model. If all the segments for a pixel, rather than just the closest one, are considered, the Ray Casting Machine can calculate the volume of the object or other geometric quantities about the CSG object. A prototype Ray Casting Machine completed in 1988 computes the surfaces of CSG objects with 32 primitives in near real time, although shading calculations, which are currently performed by the host computer, take several seconds.

18.9.3 Object Parallelism versus Image Parallelism

Although object-parallel architectures are simple and appealing, they have received much less attention than have the image-parallel techniques discussed previously; a number of experimental object-parallel systems have been proposed, but few have led to commercial products. Several factors may be responsible:

- Object-parallel systems typically require specialized processors. This implies heavy reliance on custom VLSI chips, making system design difficult and expensive. Image-parallel systems, on the other hand, place more reliance on frame-buffer memory, which can be built with commercial parts such as VRAMs.
- The specialized nature of object processors limits the types of primitives that can be displayed and the shading algorithms that can be used.

- Object-parallel systems have poor overload characteristics. Generally, object-parallel systems perform at full speed as long as there are enough object processors. Special provisions must be made to handle large databases, and performance generally decreases rapidly.

In addition to these factors, system designers have so far been able to increase system performance using image parallelism alone, so the designers of commercial systems by and large have not had to confront the challenges of object-parallel systems. Nevertheless, as discussed in Section 18.8, image parallelism may be reaching a point of diminishing returns, and object parallelism may appeal to the designers of future systems. The following section discusses another promising approach toward increasing system efficiency and performance: building hybrid-parallel systems that combine aspects of the approaches we have discussed so far.

18.10 HYBRID-PARALLEL RASTERIZATION

We have seen that when image parallelism and object parallelism are pushed to the extreme, systems with low utilization result. This poor utilization can be reduced if object-order and image-order rasterization techniques are used in combination. Such hybrid-parallel systems frequently are more complex than are the systems we have discussed so far, but they can be much more efficient. They also provide us with one more layer of parallelism that can be used to build still higher-performance systems.

18.10.1 Virtual Buffers and Virtual Processors

A major drawback of highly parallel architecture designs, both object-parallel and image-parallel, is the low utilization of each PE. As discussed in Section 18.8.2, image-parallel architectures with many partitions can have extremely poor PE utilization. For example, in a fully instantiated processor-per-pixel system such as Pixel-Planes 4, PEs may be doing useful work less than 1 percent of the time. Similarly, in an object-parallel architecture such as Cohen and Demetrescu's processor-per-polygon pipeline, PEs may be actively rasterizing their polygons less than 1 percent of the time.

One way to achieve higher utilization is to build only a fraction of the hardware, but to allocate its resources dynamically around the screen as they are needed. Two variants of this technique exist: *virtual buffers* (for image-parallel systems) and *virtual processors* (for object-parallel systems). Like classic virtual memory, both attempt to increase a system's apparent physical resources by reallocating resources dynamically as needed.

Virtual buffers. In a virtual-buffer system [GHAR89], the screen is divided (conceptually) into a number of regions of uniform size, and a parallel rasterization buffer the size of a region computes the image one region at a time. Since this buffer is small, it can be built using fast or custom processing/memory at a reasonable price. A full-sized conventional frame buffer is generally provided to store the final image.

A region can be a single scan line, a horizontal or vertical band of pixels, or a rectangular area. Virtual buffers differ from interleaved-memory "footprint" processors in one critical respect: A virtual buffer remains in one screen region until *all* the primitives for

that region are processed, whereas an interleaved-memory footprint processor moves about the image to rasterize each primitive, typically returning to the same region more than once.

Virtual processors. A virtual-processor system is outwardly similar to a scan-line virtual-buffer system; in both cases, the image is computed one scan line at a time. A virtual-processor system, however, uses object-parallel PEs, rather than image-parallel PEs. Since object processors are needed for only the primitives active on a single scan line, the number of object processors can be a small fraction of the number of primitives. General Electric's NASA II and Deering's Triangle Processor, discussed in Section 18.9, are both virtual-processor systems.

Complete versus incremental bucket sorting. Because virtual-buffer/virtual-processor systems visit a region only once, bucket sorting is required. The shape of the region influences the nature of the bucket sort. In virtual-buffer systems with rectangular regions, primitives can be stored in the buckets of all the regions in which they appear, since most primitives lie entirely within one region. We call this *complete bucket sorting*. In systems with scan-line–sized regions, complete bucket sorting is impractical, since a single primitive typically covers more than one scan line. *Incremental bucket sorting*, in which a primitive is stored only in the bucket associated with its initial scan line, is generally the method of choice.

The virtual-buffer/virtual-processor approach has several advantages and disadvantages. The most important advantage is that it makes possible the rasterization speed of a fully instantiated image-parallel system with a fraction of the hardware. It also decreases traffic to and from the frame buffer, since each region's pixels are written only once. Finally, no full-screen z-buffer is needed.

Virtual-buffer/virtual-processor systems have two major disadvantages, however. First, extra memory is required to buffer the scene during bucket sorting. If transformed primitives require the same amount of storage as do primitives in object coordinates, this approximately doubles the memory requirements in the front end. It also places a hard limit on the number of primitives that can be displayed in a single frame. These disadvantages offset, to some degree, the advantage of not needing a full-screen z-buffer. Another important disadvantage is that latency is added to the display process. Even though bucket sorting can be pipelined with rasterization, bucket sorting for one frame must be completed before rasterization of that frame can begin. This increases the system's latency by one frame time, which can be detrimental in real-time systems.

Systolic Array Graphics Engine (SAGE). We now consider a system that uses the virtual-buffer approach (we discussed several virtual-processor systems in Section 18.9). SAGE is a scan-line virtual-buffer system that uses a 1D array of pixel processors implemented in VLSI [GHAR88]. Like other scan-line systems, it generates the image one scan line at a time. However, SAGE uses an image-parallel z-buffer algorithm to rasterize primitives within a single scan line.

In addition to the array of pixel processors, SAGE contains auxiliary processors to maintain the active-edge list and to break polygons into spans (see Fig. 18.25). A *polygon manager* maintains the list of active polygons as scan lines are processed (at each new scan line, it adds polygons from the next bucket and deletes polygons that are no longer active).

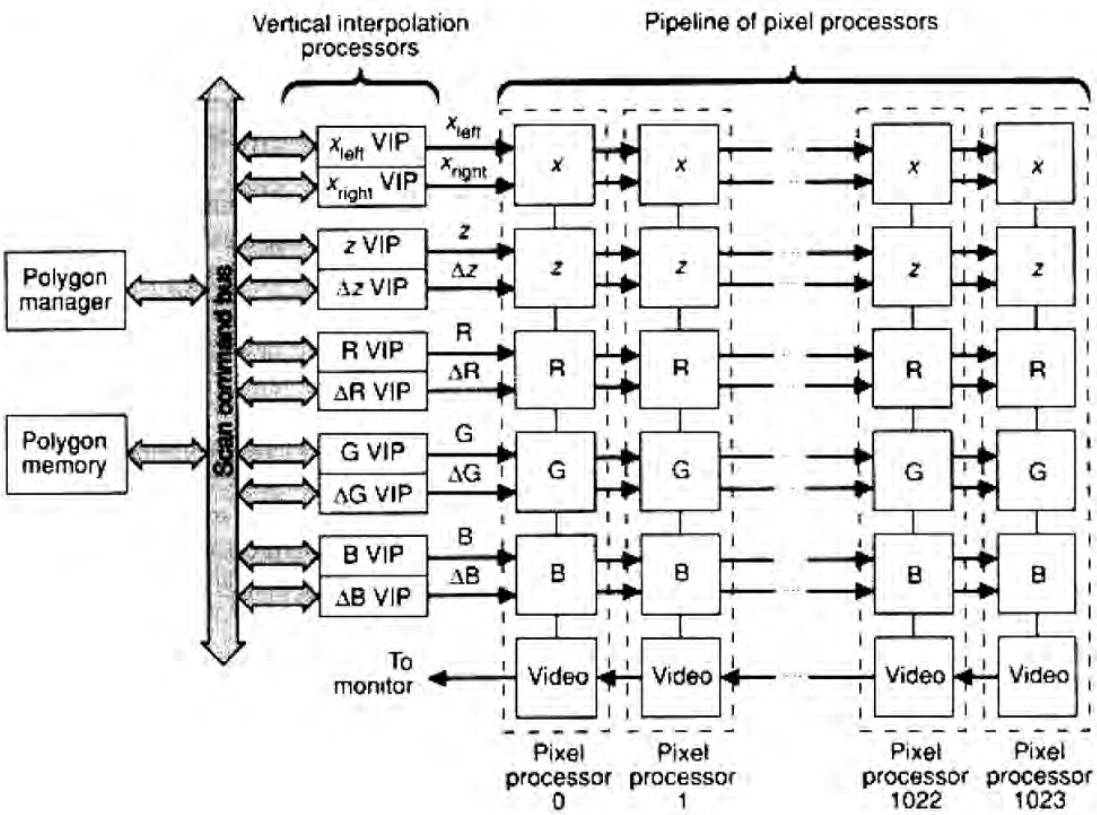


Fig. 18.25 Block diagram of a complete SAGE system.

It also calculates delta values for computing span endpoints on successive scan lines. *Vertical interpolation processors* use these delta values to compute span endpoints for each polygon active on the current scan line. So that many accesses to polygon memory can be avoided, these processors contain internal memory for storing delta values and x, z, R, G, and B values for up to 512 active polygons.

After these set-up operations, the vertical interpolation processors load the endpoints of each active polygon's current span into the pipeline of pixel processors. When a pixel processor receives a span, it compares its local x value against the span's endpoint values. If the pixel lies within the span, the processor calculates z and color values for its pixel (again using the span-endpoint values). If the new z value is smaller than the value stored at the pixel, the z and color values for the pixel are updated. Pixel values are scanned out for display using a serial shifting scheme similar to that used in VRAMs. The video stream flows in the opposite direction from that of the stream of spans.

18.10.2 Parallel Virtual-Buffer Architectures

Another level of parallelism is available in virtual-buffer systems that use rectangular regions and complete bucket sorting. Since complete bucket sorting allows a region to be rasterized without any knowledge of preceding regions (unlike incremental bucket sorting, which requires knowledge of the active-polygon list from the preceding region), a system

that uses complete bucket sorting can have multiple rasterization buffers. These rasterization buffers can work in parallel on different screen regions, allowing even faster systems to be built.

In such a system, each buffer is initially assigned to a region. When it has processed all the primitives in the region's bucket, it is assigned to the next region needing processing. This provides an additional coarse-grained level of image parallelism that can be used regardless of the rasterization method used by the individual virtual buffers.

Pixel-Planes 5. Pixel-Planes 5, which was under construction in 1989 [FUCH89], is one of the first systems to use parallel virtual buffers. Pixel-Planes 5 uses 10 to 16 logic-enhanced memory rasterization buffers that are 128 pixels on a side. Each buffer is a miniature Pixel-Planes 4 array, and is capable of rasterizing all primitives falling into a 128 by 128 region. These rasterizers can be assigned dynamically to any of the 80 regions in a 1280- by 1024-pixel screen.

Rasterization in Pixel-Planes 5 occurs in two phases. First, the front-end processors (16 to 32 floating-point processors) transform and sort primitives into screen regions. Sorting is done by checking a primitive's screen extent against the 128-pixel–aligned region boundaries; primitives that cross region boundaries are placed in the buckets of all regions affected (this occurs approximately 20 percent of the time for 100-pixel triangles). After bucket sorting is complete, the multiple rasterizers process regions in parallel. When a rasterizer finishes a region, it transfers its newly computed image to the appropriate part of a conventional frame buffer and begins processing primitives from an unassigned region (see Fig. 18.26). All communication between system components is performed over a 1.28-gigabyte-per-second token ring.

Pixel-Planes 5 achieves much higher processor utilization than did previous Pixel-Planes systems, since 128 by 128 regions are much closer to the size of most primitives than is a single 512 by 512 region. Since virtual-buffer rasterizers can be assigned to screen regions dynamically, system resources can be concentrated on the regions that need them the most.

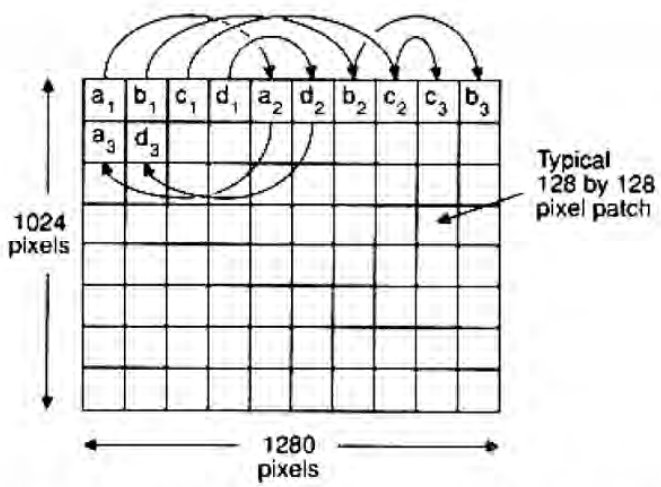


Fig. 18.26 Dynamic assignment of rasterizers to regions in Pixel-Planes 5. (Reprinted with permission from [FUCH89], © ACM.)

Parallel virtual-buffer systems do present two difficulties. First, transferring buckets to multiple rasterization buffers in parallel requires a high-performance data network between the front-end and rasterization subsystems, as well as sophisticated control and synchronization software [ELLS89]. Second, in some images, most of the primitives in the database can fall into a single region, making the extra layer of parallelism useless. The primitives in the overcrowded region could be allocated to more than one rasterizer, but then the multiple partial images would have to be combined. Although this complicates the rasterization process, it can be done by compositing the multiple images into one buffer at the end of rasterization.

18.10.3 Image-Composition Architectures

The notion of combining images after rasterization can be used to build a second type of multilevel parallel architecture, *image-composition* or *composite* architectures [MOLN88; SHAW88]. The central idea is to distribute primitives over a number of complete rendering systems. The multiple renderers are synchronized so they use identical transformation matrices and compute the same frame at the same time. Each renderer then computes its partial image independently and stores that partial image in its own frame buffer.

Video scanout from each frame buffer occurs in the normal way, except that z-buffer contents are scanned out as well. Scanout processes in each frame buffer are synchronized so that each frame buffer scans out the same pixel at the same time. A tree or pipeline of compositors combines the RGB and z streams from each renderer using the technique

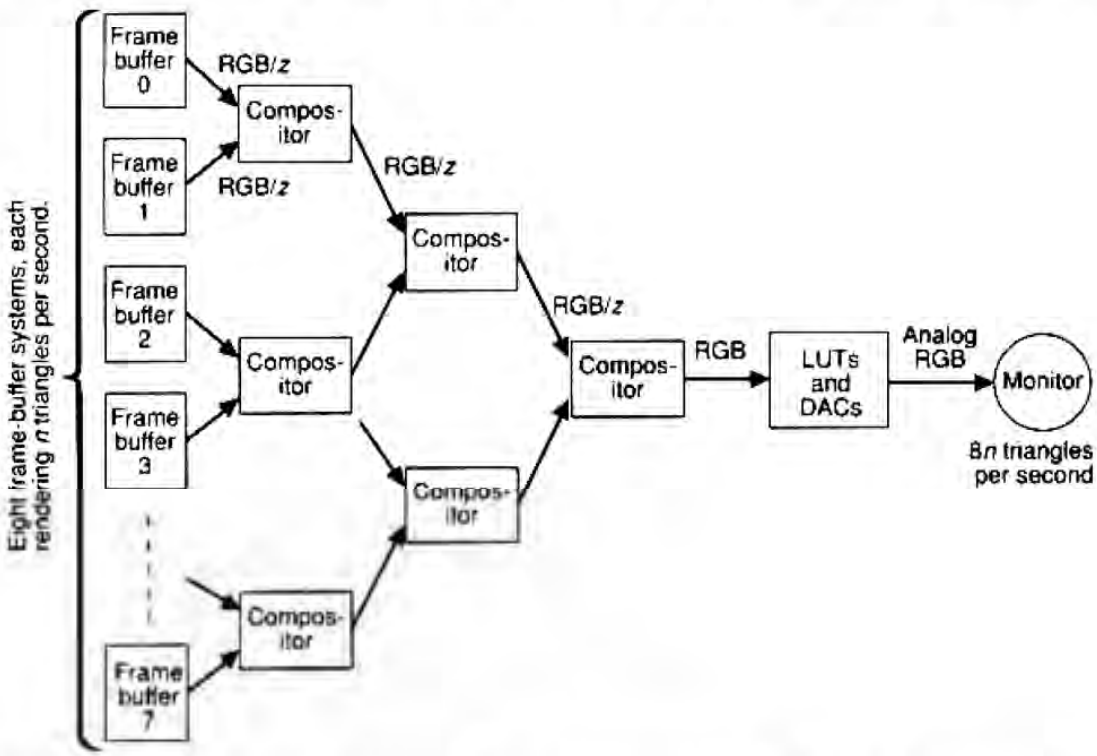


Fig. 18.27 An image-composition system composed of eight individual renderers.

described in Section 18.9.2. Figure 18.27 shows a composite system for displaying $8n$ triangles per second built from eight renderers, each of which can display n triangles per second.

This technique can be exploited to build systems of arbitrarily high performance by using a large number of parallel renderers. The main difficulties with this approach are the need to distribute the database over multiple processors, which incurs all the difficulties of parallel front ends (described in Section 18.6.1); aliasing or erroneous pixels caused by the image-composition operation; and a lack of flexibility, since image composition restricts the class of rasterization algorithms that can be implemented on the machine. Nevertheless, this approach provides an important way to realize systems of extremely high performance.

18.11 ENHANCED DISPLAY CAPABILITIES

This section discusses the architectural implications of a variety of enhancements to the standard Gouraud-shaded polygon-rendering algorithm. Increasing numbers of these features are judged necessary or desirable by suppliers and users. At the end of this section, we discuss a few aspects of flight simulators, the systems that traditionally have included the greatest number of advanced features.

18.11.1 Support for Multiple Windows

As mentioned in Chapter 10, the ability to display multiple overlapping windows controlled by separate applications has become a necessity in current workstations. Without hardware support for this capability, the overall speed of the graphics system may suffer seriously. For example, operations such as pushing one window behind another, popping a window so that all of it becomes visible, and dragging a window around the screen, all require the ability to copy pixels rapidly from one frame-buffer region to another, and, in some cases, to regenerate part or all of an image. Also, if multiple images are to be generated in different windows simultaneously, the rendering pipeline must be able to switch rapidly between applications.

The architectural enhancements needed to provide such support often cut across the design of the entire graphics subsystem: from display traversal, to geometric transformation, to clipping, to rasterization, and to the frame-buffer memory organization. We discuss here some of the main difficulties in building a windowing system that supports multiple interactive applications. For a more complete treatment of these issues, see [RHOD89].

Context switching between applications. When the graphics system switches from an application running in one window to an application running in another, state information must be saved and restored at many places in the graphics pipeline. For example, in the modeling-transformation stage, the stack of current transformation matrices must be saved for the current application and loaded for the next application; in the video controller, look-up table entries may have to be reassigned. To save and restore state rapidly, local memory is needed at many stages in the rendering pipeline.

Having a fixed amount of local storage, of course, limits the number of applications that can run simultaneously. If fast context switching is not possible, graphics systems

generally switch between windows as infrequently as possible (perhaps only a few times per second), causing a noticeable hesitation in the movement of objects in the various windows.

Clipping to nonconvex screen windows. When windows overlap, pixels must be written only to the exposed portions of their respective windows. This requires clipping pixels to window boundaries that may not be convex. Several solutions to this problem are possible.

Writing an entire window to offscreen memory. This allows the rendering pipeline to be largely isolated from windowing issues. A fast pixel copy can transfer exposed portions of the window to the actual frame buffer after each frame update. This requires extra processing for the pixel copy and adds latency to the image-generation process, both of which can seriously hamper interactivity. Alternatively, the video system can assemble the video stream on the fly from disjoint portions of video memory, as is done in Lexidata's Lex 90 system [SUYD86] or Intel's 82786 graphics coprocessor chip [SHIR86]. This approach does not add latency, but may restrict the locations and number of windows, since assembling the video image on the fly from different memory locations places heavy demands on the memory system.

Selectively updating screen pixels. A second alternative is to compute all the pixels for each window, but to update only those pixels in exposed portions of the window. This is analogous to scissoring, described in Chapter 3. To make this operation fast, hardware support is generally needed. A popular approach is to use window ID bits [AKEL88]. Here, a few (generally 4 to 8) additional bits are associated with each pixel. These bits specify a unique ID for each window. When the frame buffer is updated, hardware compares the window ID of the pixel to be written with the window ID stored in the frame buffer. The frame-buffer pixel is updated only if the window IDs agree.

Clipping within the rendering pipeline. The main disadvantage of both of the preceding schemes is that they generate many pixels that may never be displayed. The alternative is to clip within the rendering pipeline, so that the only pixels computed are those that will actually be displayed. The main difficulty with this approach is that the exposed part of a window may not be a simple rectangle. It may have concavities caused by partially overlapping windows of higher priority, or it may even contain a hole consisting of a smaller, higher-priority window. Clipping to arbitrary boundaries such as these is computationally expensive. One solution is to tesselate the exposed region with a series of rectangular windows and to clip primitives to these simple window boundaries. Of course, this too affects system performance, since primitives must be processed multiple times, once for each rectangular region.

It is extremely difficult, in general, to provide full performance for all possible window configurations and desired actions. System designers typically begin with a list of desired windowing capabilities and try to find ways of implementing these without incurring exorbitant system costs. Most systems run at full speed only under certain simple situations, and take a big performance hit otherwise. For example, a system may run at full speed only when the rendering window is fully uncovered (or is mostly uncovered, or when its visible portion is a rectangular region), or its rendering speed may suffer when more than one window is being updated simultaneously.

18.11.2 Support for Increased Realism

Our discussion of graphics architecture thus far has primarily concerned the quest for speed. With the continued increase in speed of affordable systems over the past years, users have desired increasingly realistic images as well. Algorithms for generating realistic images are discussed in Chapters 14, 15, 16, and 20; we discuss here the impact realistic rendering has on graphics system architecture.

Antialiasing. As discussed in Chapter 14, the two main approaches to antialiasing use area sampling and point sampling. In area sampling, we need to know the fractional contribution of each primitive to each pixel (ideally, we would like to weight these contributions using an appropriate filter). Unfortunately, calculating the precise fractional contribution is very difficult for 3D images. Architectures that provide hardware support for antialiasing generally use some sort of supersampling scheme. Depending on the characteristics of the rasterization processors, uniform or adaptive supersampling is used.

Uniform supersampling. Uniform supersampling involves calculating the entire image at high resolution and combining multiple sample points at each pixel to determine the pixel's single color value. This appears to be the only feasible approach in SIMD image-parallel systems. Unfortunately, supersampling with n samples per pixel reduces system performance by a factor of n. For this reason, some systems of this type adopt a successive-refinement approach to antialiasing, in which a crude image is generated at full speed when high update rates are desired; when no changes in the image are pending, the system computes further sample points and uses them to refine the image [FUCH85].

Adaptive supersampling. Adaptive supersampling involves calculating the image at high resolution only where necessary. This can be performed most readily on image-order rasterization systems, where information is available on whether or not pixels are partially covered. A well-known adaptive supersampling technique useful for object-order rasterization is the A-buffer (see Section 15.7.3). The linked lists needed to store partially covering primitives are feasible only in MIMD systems that have an arbitrary amount of memory available for each pixel.

Transparency. As mentioned in Chapter 16, true transparency, including refractions and translucency, is difficult to model in software. Even simple models of transparency that exclude refraction and translucency are difficult to implement on many hardware systems. Image-parallel architectures in particular have difficulty with transparency, since the z-buffer algorithm does not handle multiple surfaces well (an extra z value is potentially needed for every transparent surface). Image-parallel systems with SIMD PEs, or systems with fixed amounts of storage per pixel, can generally display a few transparent objects by rasterizing in multiple passes—one pass for all the opaque primitives, and then a separate pass for each transparent surface [MAMM89]. If more than a few transparent surfaces are required, a screen-door transparency scheme generally must be used. MIMD systems may be able to represent pixels as linked lists of visible surfaces, allowing larger numbers of transparent objects to be displayed.

Object-parallel architectures handle transparency more gracefully, since they do not require intermediate storage for the entire image, and have all the potentially visible

polygons in hand when the color for each pixel is computed. The rasterizer, however, must be configured to allow multiple covering polygons and to perform the weighted-sum calculations needed to compute final pixel values.

Textures. Until recently, the only systems that performed texturing at interactive rates were multimillion-dollar flight simulators. Increasingly, however, different forms of texturing are becoming available on graphics workstations and even low-end systems. Textures can be simulated, of course, on any architecture using myriad small triangles with interpolated shading between vertices. This approach, however, places high demands on the display system and is feasible only in high-performance workstations or when simple textures are displayed.

The two traditional methods of texturing are texture maps and procedural textures (see Chapter 16). The drawback of both of these methods is that texturing is applied at the end of the rendering pipeline — after the image has already been converted into pixels. In many architectures, pixel data are not accessible by a general-purpose processor at this point. Flight simulators add an extra hardware stage to perform texture-map lookups. A few graphics workstations, such as the Stellar GS2000 (see Section 18.11.3), store the generated image in main memory, where it is accessible by the system's general-purpose processors.

In architectures in which large-grain MIMD processors have access to the computed image, the texture-map approach is the most appropriate. Texture coordinates, rather than colors, can be stored at each pixel. If each of these MIMD processors is provided with a copy of the texture maps, it can perform the texture-map lookup, replacing the texture coordinates with the appropriate colors. Note that simple texture-map lookup is generally insufficient here — some sort of multiresolution texture map or summed-area table is needed to avoid aliasing.

In architectures in which only fine-grain parallel processors have access to the computed image, procedural textures may be more feasible. Fine-grain processors may not contain enough memory to store the texture maps. If many PEs are available, however, they may be able to compute simple procedural texture models at interactive rates [FUCH89].

Shadows. Computing true shadows, as described in Section 16.4, exacts a high computational price; on many architectures, true shadows cannot be computed at all. For these reasons, few high-performance systems implement true shadows [FUCH85], although a number of techniques can be used to approximate shadows under certain conditions. A common technique used in commercial flight simulators and in a number of workstation systems is to generate shadows for the ground plane only, as described in Section 16.4.

Ray-tracing architectures. Ray tracing, described in Chapters 15 and 16, is a powerful rendering method that can generate extremely realistic images. Unfortunately, it requires a great deal of computation (a typical image can require minutes or hours to compute on a typical workstation). Fortunately, ray-tracing algorithms can be parallelized in several ways, many of which have analogs in conventional rendering:

- *Component parallelism.* Computations for a single ray can be parallelized. For example, reflection, refraction, and intersection calculations all require computing the

x, y, and z components of vectors or points. These three components can be calculated in parallel, resulting in a speedup by a factor of 3.

- *Image parallelism*. Ray–primitive intersections can be calculated in parallel in separate PEs, since the calculations for each ray are independent. To take advantage of this form of parallelism, however, PEs potentially need access to the entire database, since the ray tree for a particular ray may reach any portion of the database.
- *Object parallelism*. Primitives in the database can be distributed spatially over multiple PEs. Each PE, then, is responsible for all rays that pass through its region. It computes ray–object intersections if the ray hits an object, and forwards the ray to the next PE otherwise.

Many architectures for ray tracing have been designed using these techniques alone or in combination. Most use a large number of MIMD PEs, since ray-tracing typically requires a large amount of branching and random addressing.

The simplest type of image-parallel architecture assigns one or more rays to each PE and replicates the entire database at each PE. This technique was used in the LINKS-1 [NISH83a], built at Osaka University in 1983. The LINKS-1 has been used to compute numerous animation sequences. This technique was also used in SIGHT [NARU87], a more recent design done at Nippon Telegraph and Telephone. The SIGHT architecture also takes advantage of component parallelism in its TARAI floating-point unit.

The first proposed object-parallel ray-tracing architectures used uniform spatial subdivision to assign portions of the universe to PEs [CLEA83]. This resulted in poor efficiency for many scenes, since most of the primitives were clustered in a few regions. Other proposed architectures subdivided the scene adaptively to improve the load balance among PEs [DIPP84]. Although this approach improves PE utilization, it makes mapping rays to PEs much more difficult.

Since many ray-tracing architectures closely resemble commercial parallel computers, which offer advantages of lower cost and mature programming environments, research in parallel ray tracing has shifted largely to developing efficient algorithms for commercial multiprocessors, such as hypercubes, Transputer meshes, and the Connection Machine. Two chief concerns are achieving high utilization and balanced load among PEs, particularly when distributed database models are used.

An image-parallel ray-tracing algorithm has been developed for Thinking Machines' SIMD Connection Machine [DELA88], in which the database is repeatedly broadcast to all of the PEs, which perform ray–object intersections in parallel.

Implementations have also been reported on shared-memory multiprocessors, such as the BBN Butterfly [JENK89]. Here, the database does not need to be stored at each PE or broadcast repeatedly; instead, PEs request portions of the database from the memory system as needed. Unfortunately, contention for shared memory resources increases with the number of PEs, so only modest performance increases can be achieved in such systems.

Nemoto and Omachi [NEMO86], Kobayashi and Nakamura [KOBA87], Scherson and Caspary [SCHE88], and others (see [JEVA89]) have proposed various methods for adaptively assigning objects to PEs and for passing rays between those PEs. AT&T's Pixel Machine, a MIMD multiprocessor with PEs based on digital signal-processor chips, ray

traces simple images at interactive rates [POTM89]. Such systems offer a dramatic glimpse of the performance possible in parallel ray tracers of the future.

18.11.3 Stellar's GS2000—A Tightly Integrated Architecture that Facilitates Realistic Rendering

Systems such as Silicon Graphics' POWER IRIS (Section 18.8.2) use dedicated hardware to compute images rapidly. Many of these calculations, particularly in the geometric transformation and other front-end stages, are similar to the types of processing needed in many types of scientific computing. The Stellar GS2000 (similar to its predecessor, the Stellar GS1000 [APGA88]) seeks to make its computing resources available both for image generation and for accelerating other compute-intensive jobs that may run on the machine.

Figure 18.28 shows a block diagram of the GS2000 architecture. The Multi-Stream Processor is a single high-performance processor that simultaneously executes instructions from four independent instruction streams; its peak processing rate is 25 million instructions per second. The vector floating-point processor performs scalar and vector floating-point operations; it can process a maximum of 40 million floating-point operations per second. The rendering processor uses a setup processor to perform polygon-processing calculations and a 4 × 4 SIMD footprint processor to perform pixel operations. The GS2000 renders 150,000 pseudocolored, Gouraud-shaded, z-buffered, 100-pixel triangles per second and 30,000 Phong-shaded polygons per second.

All the GS2000's processing and memory resources are organized around a central 512-bit-wide communication structure called the DataPath. The DataPath has an unusual design in which all of the I/O connections and registers for a single bit of the datapath are built onto a custom gate-array chip (each DataPath chip contains circuitry for 16 bits, so a total of 32 are needed).

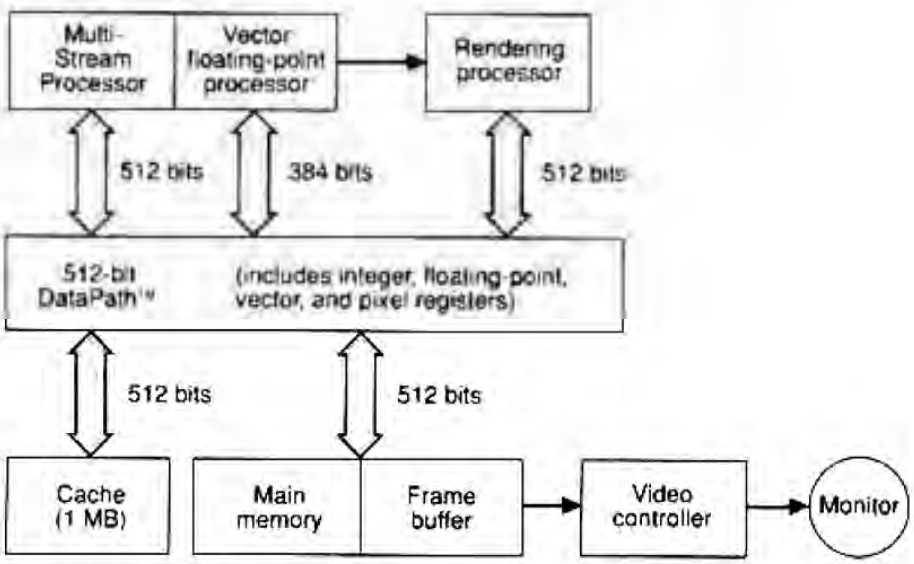


Fig. 18.28 Block diagram of the Stellar GS2000 (based on [APGA88]).

In a typical graphics application, the Multi-Stream Processor traverses a structure database stored in main memory; it then feeds data to the vector processor, which performs geometry calculations. The vector processor sends transformed primitives to the rendering processor, which calculates the pixels of the image. Rather than storing images directly in a frame buffer, the rendering processor stores them in main memory as *virtual pixel maps*, visible portions of which are copied to the appropriate portion of the frame buffer for display.

Because virtual pixel maps are stored in main memory and are accessible to the Multi-Stream Processor and vector processor, a variety of postprocessing operations (as well as image processing and other operations) can be performed. For example, to display textures, the rasterizing unit generates texture indices in pixels, rather than final colors. Later, one of the general-purpose processors passes over the half-generated image and substitutes proper color values for each pixel by table lookup of texture indices. Also, overlapping windows can be implemented easily, since the contents of each screen window are always available in main memory. The main disadvantages of virtual pixel maps are the extra bandwidth and time required to copy image data from main memory to the frame buffer.

18.11.4 Support for Advanced Primitives

We have focused so far on architectures that display polygons rapidly. Increasing demand is being placed on systems to handle other types of primitives. Many complex-surface primitives, such as spline patches and mesh primitives, can be converted into polygons with little time penalty. For other types of primitives, however, such conversions can be either time-consuming or difficult. For example, converting a CSG object to boundary representation is time-consuming, and can result in a large expansion in the amount of data; polygonalizing a volume dataset is slow as well, and tends to obscure data and to produce undesirable artifacts. When more complicated primitives are displayed directly, many of these difficulties can be avoided; in many cases, system performance can be increased as well.

Curved surfaces. As discussed in Chapter 11, polygons are simple, regular primitives that are convenient to display (especially in hardware). Unfortunately, since they are very low-level, they are inconvenient for modeling and lead to inaccuracies, especially for complex, curved surfaces. Other representations for surfaces, such as Bezier patches and NURBS, are more convenient to specify when modeling, but require complicated processing to render directly.

Most hardware systems that display high-order primitives decompose the primitives into polygons for display purposes. For many types of surface primitives (particularly Bezier patches and NURBS), this decomposition can be done rapidly in hardware using forward-difference engines [LIEN87]. In some cases, however, it may be faster to display curved surfaces directly, especially in systems with programmable parallel rasterization processors. When this book was written, few systems provided this capability [MCLE88]. It remains to be seen whether systems of the future will continue to tile such surfaces with polygons, or whether they will render curved surfaces directly.

Volume data. Volume rendering—the direct rendering of data represented as 3D scalar fields (discussed in Section 20.6)—is becoming an important branch of computer graphics. In many ways, it has an even greater need for hardware acceleration than does polygon rendering, since volume datasets are generally much larger than are polygon datasets (a typical dataset from a CT scanner might be $64 \cdot 256 \cdot 256 = 4.2$ million voxels, whereas few polygon datasets contain more than 1 million polygons). Furthermore, voxel calculations are simpler than polygon calculations.

The first architectures designed to accelerate the display of volume datasets classified voxels as either "occupied" or "empty," and displayed occupied voxels. This approach minimizes the amount of processing but obscures data in the interior of the object and produces a "sugar-cube" effect in the generated image. Phoenix Data Systems' Insight system [MEAG85] uses this approach: It interactively displays volume datasets encoded as octrees. Kaufman's Cube architecture [KAUF88b] provides special hardware for accessing in parallel the entire row of voxels corresponding to a single pixel. In addition, this hardware can determine the nearest visible voxel in the row in logarithmic time. Views from arbitrary angles are implemented by rotating the dataset in Cube's 3D memory.

More recent volume-rendering methods assign partial transparencies (or opacities) to each voxel, rather than using binary classification, and attempt to eliminate sampling artifacts (see Section 20.6). Since more voxels contribute to each pixel than when binary classification is used, these algorithms substantially increase image-generation time.

Volume-rendering architectures based on these algorithms generally use variations of the image- or object-parallel approaches we have seen before. Image-parallel architectures assign PEs to pixels or to groups of pixels [LEVO89]. Object-parallel architectures generally partition the voxels into contiguous 3D regions and assign to each a separate PE [WEST89]. During rendering, PEs compute their voxels' contributions to each pixel in the image as though their voxels were the only ones in the dataset. The contributions from multiple regions are then composited using these aggregate colors and opacities.

The primary advantage of the object-parallel approach is that it divides the database among PEs. The disadvantage is that pixel values calculated in one PE may be obscured by pixels calculated in another PE whose voxels lie nearer to the eyepoint. Image parallelism has complementary advantages and disadvantages: Each PE processes only voxels that are potentially visible, but each PE needs access to the entire (very large) database.

Dynamic Digital Displays' parallel Voxel Processor system is an example of an object-parallel architecture [GOLD88]. A hypercube architecture could also be used in an object-parallel approach by assigning each voxel in each image slice (or even in the entire database) to a separate PE. Levoy proposed a hybrid system using Pixel-Planes 5, in which voxel shading is done in an object-parallel fashion and the image is generated in an image-parallel fashion [LEVO89]. The Pixar Image Computer computes red, green, and blue components of an image in parallel [LEVI84]. Because volume rendering is a relatively new area, it is difficult to predict what architectures (or even algorithms) will be dominant in the future.

Constructive-solid-geometry (CSG) architectures. As we saw in Chapter 12, CSG is one of the more popular techniques for modeling solid objects. Chapter 15 described ways to display a CSG object directly from that object's binary-tree representation. The two

popular approaches for direct CSG rendering are generalizations of the image-order and object-order rasterization techniques described earlier in this chapter.

Image-order CSG rasterization. This approach was described in Section 18.9.2. Kedem and Ellis's Ray Casting Machine is a hardware implementation of this algorithm. Their current system displays small CSG objects at interactive rates.

Object-order CSG rasterization. Chapter 15 also described object-order (or depth-buffer) CSG rasterization algorithms that are generalizations of the z-buffer algorithm. CSG depth-buffer algorithms can be implemented on any architecture that has enough memory for each pixel to store two z-buffers, two color buffers, and three 1-bit flags [GOLD86]. Many current high-performance workstations provide frame buffers with sufficient memory. If enough pixel-level processing is available, depth-buffer CSG display algorithms can display modest objects at interactive speeds [GOLD89]. If more memory per pixel is available, more complicated CSG objects can be displayed even faster [JANS87].

18.11.5 Support for Enhanced 3D Perception

Recall from Chapter 14 that 3D images displayed on a 2D screen contain only a few 3D depth cues: obscuration, kinetic depth effect, lighting, and occasionally shadows. The real 3D world provides such additional powerful cues as stereopsis and head-motion parallax. This section discusses architectures and architectural enhancements that seek to provide these extra cues.

Stereo display. Stereopsis can be achieved with a 2D display by computing separate images for the left and right eyes and channeling each image to the respective eye. Image pairs can be displayed on separate monitors (or in separate windows on the same monitor), or left and right images can be displayed in alternating frames. In the former case, image pairs can be viewed with a stereo viewer that optically channels the two images to a fixed point in front of the screen where the viewer must be positioned. Disadvantages of this scheme are that only one person at a time can view the scene, and that only one-half of the monitor's resolution is available (unless two monitors are used, in which case the cost of the system increases dramatically).

Multiplexing left and right images in time is a more popular technique. This requires displaying left and right images alternately in rapid succession, and blocking each eye's view while the other eye's image is being displayed. A graphics system displaying stereo images generally must have a frame buffer large enough to store four complete images—enough to double-buffer the image for each eye. Also, some scheme is needed to block each eye's view of the screen at the appropriate time. One approach uses a mechanical shutter synchronized to the frame buffer so that left and right images are displayed at the correct time [LIPS79]. Unfortunately, mechanical shutters can be heavy, noisy, and, in some cases, dangerous (one design used a rapidly spinning cylinder less than 1 inch from the eye).

A more popular mechanism is an electronic shutter that alternately polarizes light in one direction and then another. The electronic shutter may be the same size as and mounted in front of the display screen, or it can be smaller and worn on special goggles. In either case, lenses polarized in opposite directions are placed before each eye. When the polarization of the electronic shutter corresponds to the polarization of one of these lenses,

the screen becomes visible; when the polarization is in the opposite direction, the view is blocked. Placing the shutter in front of the display screen allows several users to view the image simultaneously, if each wears a pair of inexpensive, passive glasses. Large electronic shutters are expensive, however. Stereo displays with electronic shutters tend to be darker than mechanical ones as well, since the polarizing film transmits only a fraction of the light from the monitor.

Varifocal mirror. A *varifocal mirror* is an unusual display device that uses an oscillating mirror to display true 3D images. These images provide stereopsis and head-motion parallax without requiring the user to wear special headgear. The basic idea is to use a flexible mirror whose focal length can be changed rapidly, and to position it so that it reflects an image of the display monitor to the viewer (Fig. 18.29) [TRAU67; FUCH82; JOHN82]. When the mirror is vibrated with an ordinary loudspeaker, the mirror's focal length changes sinusoidally. This is generally done at a frequency of approximately 30Hz.

The mirror's periodically changing focal length makes the distance to the monitor appear to increase and decrease by several inches or more during each 30-Hz cycle. Points, lines, and similar data are displayed on a point-plotting (not raster-scan) display monitor. The perceived depth of a point is determined by its position in the display list: "near"

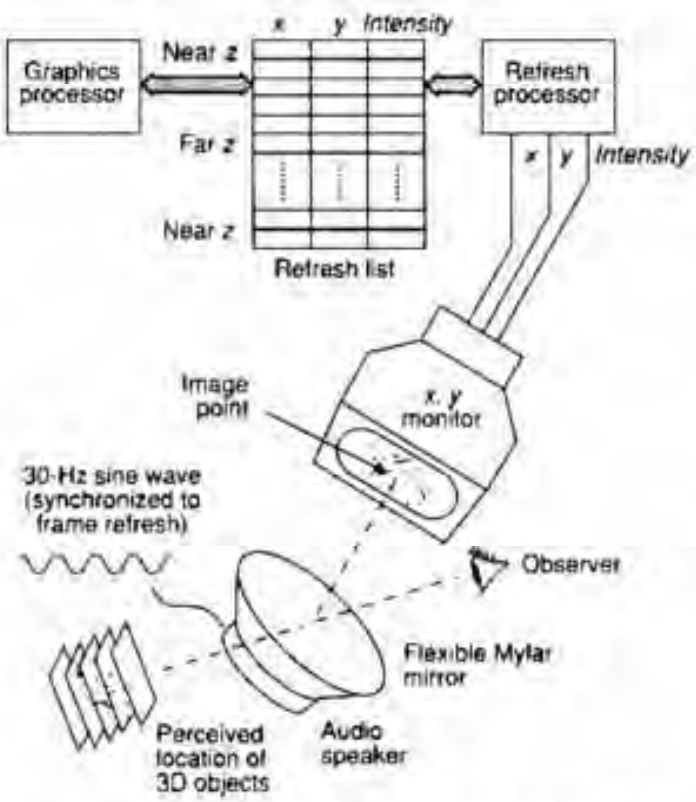


Fig. 18.29 Varifocal-mirror display system.

points are stored in the beginning or end of the refresh list; "far" points are stored in the middle of the refresh list. Note that there are two places in the list at which a point appears at the same depth—when the mirror is moving forward, and when it is moving backward.

Varifocal-mirror displays have several limitations. One drawback is that nearer objects do not obscure more distant ones, and thus only nonobscuring primitives such as points, lines, and transparent volume data can be displayed. A second difficulty is that only a limited amount of data can be displayed at a time—the amount that can be refreshed during a single mirror cycle. In spite of these limitations, the varifocal mirror is one of the very few true 3D display devices to be made into a commercial product, the Genisco SpaceGraph [STOV82].

Head-mounted display. In a seminal address at the 1965 International Federation for Information Processing Congress [SUTH65], Ivan Sutherland proposed that the *ultimate display* would be one that produced images and other sensory input with such fidelity that the observer could not tell the simulated objects from real ones. In 1968, he showed a prototype display that was worn on the head and that demonstrated the most important property of the *ultimate display*—it allowed the user to walk around in a *virtual world*. Specifically, the system comprised the following:

1. Headgear with two small display devices, each optically channeled to one eye
2. A tracking system that allowed the computer system to know the precise location of the user's helmet (and thus head) at all times
3. A hand-held wand, whose position was also tracked by the system, that allowed the user to reach out to grab and move objects in the virtual environment
4. A real-time graphics display system that constantly regenerated the images to the display devices as the user moved, giving the user the illusion of walking around "virtual" objects in the room

This system, with its rich 3D cues of head-motion parallax and stereopsis and its simple, direct 3D manipulation of objects inside the virtual world, convincingly demonstrated to many people that the conventional way of interacting with 3D scenes using a desktop CRT is an unsatisfying, constrained mechanism that keeps the user outside the CRT's window to the 3D virtual world. Unfortunately, several technical problems have prevented the head-mounted display from reaching full effectiveness—indeed, they have kept it to a level discouragingly close to the capabilities of Sutherland's 1968 prototype [FISH86; CHUN89]. These technical problems include:

1. Developing headgear with high-resolution displays that allows a wide-screen view of the graphics display screen superimposed on the real world (a difficult optical problem)
2. Developing a tracking system for the helmet and hand that has the range of a room with 1-millimeter resolution and response time of a few milliseconds or less
3. Designing a graphics system that generates at least 30 frames per second with minimal latency (low latency is a particularly important concern here, since latency in a head-mounted display can induce motion sickness [DEYO89]).

The most visible and widespread use of head-mounted displays has been for heads-up displays in cockpits of military aircraft. Heads-up displays, however, give only auxiliary information to the user, rather than creating a virtual world, complete with objects that can be directly manipulated. In 1989, two companies, Autodesk and VPL Research, introduced commercial head-mounted display systems based on off-the-shelf technology. Proponents of such systems predict that these kinds of systems will be the twenty-first century graphics equivalents of Sony Walkman personal stereos, to be used not only for real-time 3D applications but also for general portable interactive computing.

Digital holography.[4] Holography is another method for displaying true 3D images without using special headgear or tracking the viewer's location. Traditional holograms are produced by exposing photographic film simultaneously to laser light scattered from the object to be recorded and to a reference beam from the same laser. The interference patterns recorded on the film encode the object's appearance from a range of viewpoints. The hologram is viewed by illuminating it with laser light from the opposite direction.

Holograms of imaginary objects can be produced by simulating the laser-interference process on a computer and writing the computed fringes onto high-resolution film. Unfortunately, holograms produced by this technique can require 10^{12} Fourier transforms (each of which requires a large number of multiplications and additions) and need a VLSI-type electron-beam writer to inscribe the results on film [DALL80; TRIC87]. For these reasons, this technique is currently too expensive for almost all applications. Fully computed holograms, however, contain more information than is needed by the human eye. A promising approach is to reduce these calculations to the number actually needed. This is the subject of ongoing research.

Holographic stereograms, which are built up from a sequence of computer-generated perspective views, are attractive in the meantime. In a holographic stereogram, a sequence of approximately 100 views from slightly differing side-to-side viewpoints is projected with laser light onto holographic film, each from the direction from which it was calculated. A second reference beam from the same laser overlaps the projection beam to record the view direction in an interference pattern. After exposure and processing, the film (now the holographic stereogram) is illuminated by a reference beam in the opposite direction. Image beams then diffract back in the directions from which they were projected. An eye moving from one view to the next perceives a smooth progression of perspective information that yields an impression of a solid object or 3D scene floating in the vicinity of the film. Combined with all the conventional monocular depth cues of 3D computer graphics, this holographic image gives a particularly effective sense of shape and space [BENT82].

Although holographic stereograms are much less expensive to produce than are traditional holograms, they still require a large amount of computation (dozens to hundreds of images for a single stereogram). Furthermore, the need to record the stereogram on photographic film adds expense and time to the image-generation process. Consequently, digital holography is unlikely to yield interactive 3D images in the near future, although it

[4]Material for this section was contributed by Stephen A. Benton of the MIT Media Laboratory.

may prove useful for recording 3D still images, just as photographic film records 2D still images today.

18.11.6 Real-Time Flight Simulators

The systems that "put it all together" to generate the most realistic simulation of 3D scenes for interactive tasks are the multimillion-dollar flight simulators. Flight simulation is not unique in being able to benefit from truly real-time systems. It is, however, the one application for which customers have consistently been willing to spend millions of dollars for a single system, largely because of the cost and danger of training pilots solely in actual airplanes. Because the community of flight-simulator users is fairly small, and because the systems tend to use proprietary, hardware-specific software provided by the manufacturer, detailed information about flight-simulator architectures has not appeared in the literature.

Early flight simulators include General Electric's NASA II (see Section 18.9) and Evans & Sutherland designs based on the scan-line systems developed at the University of Utah in the late 1960s. Some of these early flight-simulator systems could display 1000 or more polygons in real time, but all used simple shading methods and provided few image enhancements. Later systems have not substantially increased the number of primitives that can be displayed. For example, Evans & Sutherland's current high-end system, the ESIG-1000, displays only 2300 polygons at 60 Hz [EVAN89]. Rather, system developers have increased scene realism and reduced distracting artifacts by incorporating features such as antialiasing, haze and fog, point light sources, clouds, and filtered textures [SCHA83]. The effectiveness of these techniques can be seen in Color Plates 1.5(a) and 1.5(b).

Flight simulators from major manufacturers such as Evans & Sutherland, General Electric, McDonnell-Douglas, and Singer/Link all share several architectural themes: Since flight simulation involves predictable interactions with very large datasets, these systems tend to use more specialized processing than do other graphics systems. For example, custom processors are frequently built to manage the image database, to transform primitives, to rasterize the image, and to perform image-enhancement operations afterward. A typical simulator system is composed of a long pipeline of proprietary processors [SCHA83].

Certain simplifications can sometimes be made in a flight simulator that are not possible in more general graphics systems. For example, since a typical simulator dataset involves a small number of moving objects and a wide, unchanging backdrop, the generality of the z-buffer visibility algorithm may not be needed and a simpler depth-sort algorithm may suffice.

Flight simulators also must manage complex databases without hesitation. A typical database may represent a region 100 miles square. Detail needed for low-level flight cannot be displayed when the airplane is at 40,000 feet. This requires the system to maintain object descriptions with different levels of detail that can be swapped in and out in real time. The architecture also must handle overloading gracefully, since image complexity may increase drastically at the most crucial times, such as during takeoffs, during landings, and in emergency situations [SCHU80]. Frames must be generated at least 30 times per second—even in these situations.

18.12 SUMMARY

This chapter has provided an overview of the architectural techniques used to build high-performance graphics systems. We have seen that the computational demands of many interactive applications quickly surpass the capabilities of a single processor, and that concurrent processing of various kinds is needed to meet the performance goals of demanding 3D applications, such as computer-aided design, scientific visualization, and flight-simulation.

We have shown how the two basic approaches to concurrency—pipelining and parallelism—can be applied to accelerate each stage of the display process, together with the advantages and limitations of each of these choices. The design of any real graphics system (indeed, any complex system in general) represents a myriad of compromises and tradeoffs between interrelated factors, such as performance, generality, efficiency, and cost. As a result, many real systems use combinations of architectural techniques we have described.

The current state of hardware technology also plays an important role in deciding what architectural techniques are feasible. For example, the memory-intensive architectures that now dominate the field would have been impractical just ten years ago before inexpensive DRAMs became available. In the future, we can expect rapidly improving technology to continue to improve graphics system performance.

The designs of future systems will be complicated by demands not just for rendering standard primitives such as points, lines, and Gouraud-shaded polygons, but also for rendering with more advanced capabilities—transparency, textures, global illumination, and volume data. These all complicate not only the calculations, but also the basic structure of the display process, making it more difficult to design systems with both high performance for the basic capabilities and sufficient generality to handle the advanced features.

EXERCISES

18.1 Section 5.6 showed that we can transform a plane equation by multiplying it by the transpose of the inverse point-transformation matrix. A surface-normal vector can be considered to be a plane equation in which the D component does not matter. How many multiplications and additions are needed to transform a surface-normal vector if the point-transformation matrix is composed of translations, rotations, and scales? (Hint: Consider the form of transformation matrix.)

18.2 A simple way to reduce the number of front-end calculations when displaying polygonal meshes is to use a mesh primitive, such as a triangle strip. A *triangle strip* is a sequence of three or more vertices, in which every consecutive set of three vertices defines a triangle. A triangle strip of $n + 2$ vertices, therefore, defines a connected strip of n triangles (whereas $3n$ vertices are needed to define n individual triangles). Estimate the number of additions/subtractions and multiplications/divisions required to display the sample database of Section 18.3.9 if the 10,000 triangles are contained in:

a. 5000 triangle strips, each containing 2 triangles
b. 1000 triangle strips, each containing 10 triangles
c. A single triangle strip containing 10,000 triangles.

What is the maximum speedup you could obtain in the front-end subsystem by converting a database of discrete triangles into triangle strips? Are there any disadvantages to using triangle strips?

18.3 Assume that the 10,000-triangle database described in Section 18.3.9 is displayed at 24 Hz on a pipelined graphics system with the following characteristics: 1280 by 1024 color display refreshed at 72 Hz, 32-bit color values, and 32-bit z values.

a. Estimate the data bandwidth between the following points of the display pipeline: (1) between display-traversal and modeling-transformation stages (assume that 24 32-bit words of data are required for each triangle in the object database); (2) between front-end and back-end subsystems (assume that 15 32-bit words are required for each transformed triangle); (3) between rasterizer and frame buffer; and (4) between frame buffer and video controller.

b. Repeat the calculations in part (a) for a database with 100,000 polygons with an average area of 10 pixels and the same overlap factor.

c. Repeat the calculations in part (a) for a database with 100,000 polygons with an average area of 100 pixels, but assume that only 10 percent of the pixels are initially visible.

18.4 Consider the pipelined object-order rasterization architecture described in Section 18.7.1. If separate processors are provided for polygon processing, edge processing, and span processing, all these operations can be overlapped. Assume that we have a sophisticated span processor that can process an entire pixel (i.e. compute its RGB and z values, compare z values, and update the frame buffer) in a single 250-nanosecond clock cycle. Ignoring the time required for clearing the screen between frames, calculate how many triangles per second this system can display under the following conditions:

a. 100-pixel triangles; negligible time for polygon and edge processing

b. 100-pixel triangles; 20 microseconds per triangle for polygon processing; negligible time for edge processing

c. 100-pixel triangles; 20 microseconds per triangle for polygon processing; 2 microseconds per scan line for edge processing (assume that a typical triangle covers 15 scan lines)

d. 10-pixel triangles; 20 microseconds per triangle for polygon processing; 2 microseconds per scan line for edge processing (assume that a typical triangle covers four scan lines)

e. 1000-pixel triangles; 20 microseconds per triangle for polygon processing; 2 microseconds per scan line for edge processing (assume that a typical triangle covers 50 scan lines).

18.5 A frame buffer is to be built with 32 bits per pixel, and access to the frame buffer is to be by single pixels (a 32-bit-wide memory system). What frame-buffer sizes with aspect ratios 1:1, 5:4, and 2:1 (up to a maximum dimension of 2048 pixels) are possible if the following commercial memory parts are used and no memory is to be wasted (i.e., all memory in the frame buffer should be used to store visible pixels):

a. 64K × 4 VRAMs (256 Kbit)

b. 256K × 4 VRAMs (1 Mbit)

c. 512K × 8 VRAMs (4 Mbit).

Answer the following questions assuming frame buffers of sizes 512 by 512, 1024 by 1024, and 2048 by 2048 (each refreshed at 60 Hz):

d. How frequently must the serial port of each memory chip be accessed during video scanout? (Assume that vertical and horizontal retrace times are neglible, and that VRAM outputs are not multiplexed.)

e. Given that the serial-port cycle time of the fastest VRAMs is about 35 nanoseconds, which of these frame buffers could be built? (Again, assume no multiplexing.)

f. Which frame buffers could be built if multiple pixels were read simultaneously and multiplexed as described in Section 18.1.5 (again assuming a VRAM cycle time of 35 nanoseconds)? How many pixels would have to be read at once for each of these frame buffers?

18.6 Consider the pipelined, object-order rasterization architecture described in Section 18.7.1.

a. Determine to what accuracy screen-space (x, y) vertex coordinates must be calculated (i.e., how many bits of precision are needed) if vertices are to be specified to within $\frac{1}{10}$ pixel of their true position in the following displays: a 320 by 200 PC display; a 1280 by 1024 workstation display; and a 1840 by 1035 high-definition TV display.

b. How many fractional bits are needed in left and right x slopes (calculated during polygon processing) if left and right span endpoints are to lie within $\frac{1}{10}$ pixel of their true position in all polygons displayable in each of the systems in part (a)? (Assume that vertex coordinates have been calculated with infinite precision and that additions are performed with perfect accuracy.)

c. What are the maximum and minimum possible values for Δx in each of the systems of part (a)? (Assume that horizontal edges have been recognized and removed before delta values are calculated.)

d. If fixed-point arithmetic is used, how many bits are needed to represent Δx values that can range from the minimum to maximum values calculated in part (c) and the precision calculated in part (b), for the three systems?

18.7 The performance of image-parallel architectures that partition the frame buffer into contiguous blocks of pixels is reduced if many primitives fall into more than one region. Assume that the display screen is divided into a number of regions of width W and height H and that a typical primitive covers a rectangular area of width w ($w \ll W$) and height h ($h \ll H$) on the display screen. Derive an expression in terms of W, H, w, and h for the average number of regions affected by a typical primitive, assuming that the primitive has an equal probability of appearing anywhere on the screen.