Computer Graphics

PRINCIPLES AND PRACTICE

Foley • van Dam • Feiner • Hughes

SECOND EDITION in C



THE SYSTEMS PROGRAMMING SERIES

Samsung/Google, EX1027, p. 1

Computer Graphics

PRINCIPLES AND PRACTICE

SECOND EDITION in C

James D. Foley, *Georgia Institute of Technology*
Andries van Dam, *Brown University*
Steven K. Feiner, *Columbia University*
John F. Hughes, *Brown University*

By uniquely combining current concepts and practical applications in computer graphics, four well-known authors provide here the most comprehensive, authoritative, and up-to-date coverage of the field. The important algorithms in 2D and 3D graphics are detailed for easy implementation, including a close look at the more subtle special cases. There is also a thorough presentation of the mathematical principles of geometric transformations and viewing.

In this book, the authors explore multiple perspectives on computer graphics: the user's, the application programmer's, the package implementor's, and the hardware designer's. For example, the issues of user-centered design are expertly addressed in three chapters on interaction techniques, dialogue design, and user interface software. Hardware concerns are examined in a chapter, contributed by Steven Molnar and Henry Fuchs, on advanced architectures for real-time, high-performance graphics.

The comprehensive topic coverage includes:

- Programming with SRGP, a simple but powerful raster graphics package that combines features of Apple's QuickDraw and the MIT X Window System graphics library.
- Hierarchical, geometric modeling using SPHIGS, a simplified dialect of the 3D graphics standard PHIGS.
- Raster graphics hardware and software, including both basic and advanced algorithms for scan converting and clipping lines, polygons, conics, spline curves, and text.
- Image synthesis, including visible-surface determination, illumination and shading models, image manipulation, and antialiasing.
- Techniques for photorealistic rendering, including ray tracing and radiosity methods.
- Surface modeling with parametric polynomials, including NURBS, and solid-modeling representations such as B-reps, CSG, and octrees.
- Advanced modeling techniques such as fractals, grammar-based models, particle systems, physically based modeling, and volume rendering.
- Concepts of computer animation and descriptions of state-of-the-art animation systems.

Over 100 full-color plates and over 700 figures illustrate the techniques presented in the book.

The best-selling book on computer graphics is now available in this C-language version. All code has been converted into C, and all changes through the ninth printing of the second edition have been incorporated. The book's many outstanding features continue to ensure its position as *the* standard computer graphics text and reference.

90000
9 780201 848403
ISBN 0-201-84840-6



Addison-Wesley Publishing Company

Samsung/Google, EX1027, p. 2

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`

Samsung/Google, EX1027, p. 3

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

Samsung/Google, EX1027, p. 4

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

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.

5 6 7 8 9 10-DOC-99 98 97 96

Samsung/Google, EX1027, p. 5

To Marylou, Heather, Jenn, my parents, and my teachers

Jim

To Debbie, my father, my mother in memoriam, and my children Elisa, Lori, and Katrin

Andy

To Jenni, my parents, and my teachers

Steve

To my family, my teacher Rob Kirby, and my father in memoriam

John

And to all of our students.

Samsung/Google, EX1027, p. 6

THE SYSTEMS PROGRAMMING SERIES

Communications Architecture for Distributed Systems	R.J. Cypser
An Introduction to Database Systems, Volume I, Fifth Edition	C.J. Date
An Introduction to Database Systems, Volume II	C.J. Date
Computer Graphics: Principles and Practice, Second Edition	James D. Foley Andries van Dam Steven K. Feiner John F. Hughes
Structured Programming: Theory and Practice	Richard C. Linger Harlan D. Mills Bernard I. Witt
Conceptual Structures: Information Processing in Mind and Machines	John F. Sowa

IBM EDITORIAL BOARD

Chairman:	Gene F. Hoffnagle

Samsung/Google, EX1027, p. 7

Foreword

The field of systems programming primarily grew out of the efforts of many programmers and managers whose creative energy went into producing practical, utilitarian systems programs needed by the rapidly growing computer industry. Programming was practiced as an art where each programmer invented his own solutions to problems with little guidance beyond that provided by his immediate associates. In 1968, the late Ascher Opler, then at IBM, recognized that it was necessary to bring programming knowledge together in a form that would be accessible to all systems programmers. Surveying the state of the art, he decided that enough useful material existed to justify a significant codification effort. On his recommendation, IBM decided to sponsor The Systems Programming Series as a long term project to collect, organize, and publish those principles and techniques that would have lasting value throughout the industry. Since 1968 eighteen titles have been published in the Series, of which six are currently in print.

The Series consists of an open-ended collection of text-reference books. The contents of each book represent the individual author's view of the subject area and do not necessarily reflect the views of the IBM Corporation. Each is organized for course use but is detailed enough for reference.

Representative topic areas already published, or that are contemplated to be covered by the Series, include: database systems, communication systems, graphics systems, expert systems, and programming process management. Other topic areas will be included as the systems programming discipline evolves and develops.

The Editorial Board

Samsung/Google, EX1027, p. 8

Preface

> Interactive graphics is a field whose time has come. Until recently it was an esoteric specialty involving expensive display hardware, substantial computer resources, and idiosyncratic software. In the last few years, however, it has benefited from the steady and sometimes even spectacular reduction in the hardware price/performance ratio (e.g., personal computers for home or office with their standard graphics terminals), and from the development of high-level, device-independent graphics packages that help make graphics programming rational and straightforward. Interactive graphics is now finally ready to fulfill its promise to provide us with pictorial communication and thus to become a major facilitator of man/machine interaction. (From preface, *Fundamentals of Interactive Computer Graphics*, James Foley and Andries van Dam, 1982)

This assertion that computer graphics had finally arrived was made before the revolution in computer culture sparked by Apple's Macintosh and the IBM PC and its clones. Now even preschool children are comfortable with interactive-graphics techniques, such as the desktop metaphor for window manipulation and menu and icon selection with a mouse. Graphics-based user interfaces have made productive users of neophytes, and the desk without its graphics computer is increasingly rare.

At the same time that interactive graphics has become common in user interfaces and visualization of data and objects, the rendering of 3D objects has become dramatically more realistic, as evidenced by the ubiquitous computer-generated commercials and movie special effects. Techniques that were experimental in the early eighties are now standard practice, and more remarkable "photorealistic" effects are around the corner. The simpler kinds of pseudorealism, which took hours of computer time per image in the early eighties, now are done routinely at animation rates (ten or more frames/second) on personal computers. Thus "real-time" vector displays in 1981 showed moving wire-frame objects made of tens of thousands of vectors without hidden-edge removal; in 1990 real-time raster displays can show not only the same kinds of line drawings but also moving objects composed of as many as one hundred thousand triangles rendered with Gouraud or Phong shading and specular highlights and with full hidden-surface removal. The highest-performance systems provide real-time texture mapping, antialiasing, atmospheric attenuation for fog and haze, and other advanced effects.

Graphics software standards have also advanced significantly since our first edition. The SIGGRAPH Core '79 package, on which the first edition's SGP package was based, has all but disappeared, along with direct-view storage tube and refresh vector displays. The much more powerful PHIGS package, supporting storage and editing of structure hierarchy, has become an official ANSI and ISO standard, and it is widely available for real-time

Samsung/Google, EX1027, p. 9

geometric graphics in scientific and engineering applications, along with PHIGS+, which supports lighting, shading, curves, and surfaces. Official graphics standards complement lower-level, more efficient de facto standards, such as Apple's QuickDraw, X Window System's Xlib 2D integer raster graphics package, and Silicon Graphics' GL 3D library. Also widely available are implementations of Pixar's RenderMan interface for photorealistic rendering and PostScript interpreters for hardcopy page and screen image description. Better graphics software has been used to make dramatic improvements in the "look and feel" of user interfaces, and we may expect increasing use of 3D effects, both for aesthetic reasons and for providing new metaphors for organizing and presenting, and navigating through information.

Perhaps the most important new movement in graphics is the increasing concern for modeling objects, not just for creating their pictures. Furthermore, interest is growing in describing the time-varying geometry and behavior of 3D objects. Thus graphics is increasingly concerned with simulation, animation, and a "back to physics" movement in both modeling and rendering in order to create objects that look and behave as realistically as possible.

As the tools and capabilities available become more and more sophisticated and complex, we need to be able to apply them effectively. Rendering is no longer the bottleneck. Therefore researchers are beginning to apply artificial-intelligence techniques to assist in the design of object models, in motion planning, and in the layout of effective 2D and 3D graphical presentations.

Today the frontiers of graphics are moving very rapidly, and a text that sets out to be a standard reference work must periodically be updated and expanded. This book is almost a total rewrite of the *Fundamentals of Interactive Computer Graphics*, and although this second edition contains nearly double the original 623 pages, we remain painfully aware of how much material we have been forced to omit.

Major differences from the first edition include the following:

- The vector-graphics orientation is replaced by a raster orientation.
- The simple 2D floating-point graphics package (SGP) is replaced by two packages—SRGP and SPHIGS—that reflect the two major schools of interactive graphics programming. SRGP combines features of the QuickDraw and Xlib 2D integer raster graphics packages. SPHIGS, based on PHIGS, provides the fundamental features of a 3D floating-point package with hierarchical display lists. We explain how to do applications programming in each of these packages and show how to implement the basic clipping, scan-conversion, viewing, and display list traversal algorithms that underlie these systems.
- User-interface issues are discussed at considerable length, both for 2D desktop metaphors and for 3D interaction devices.
- Coverage of modeling is expanded to include NURB (nonuniform rational B-spline) curves and surfaces, a chapter on solid modeling, and a chapter on advanced modeling techniques, such as physically based modeling, procedural models, fractals, L-grammar systems, and particle systems.
- Increased coverage of rendering includes a detailed treatment of antialiasing and greatly

Samsung/Google, EX1027, p. 10

expanded chapters on visible-surface determination, illumination, and shading, including physically based illumination models, ray tracing, and radiosity.

- Material is added on advanced raster graphics architectures and algorithms, including clipping and scan-conversion of complex primitives and simple image-processing operations, such as compositing.
- A brief introduction to animation is added.

This text can be used by those without prior background in graphics and only some background in Pascal programming, basic data structures and algorithms, computer architecture, and simple linear algebra. An appendix reviews the necessary mathematical foundations. The book covers enough material for a full-year course, but is partitioned into groups to make selective coverage possible. The reader, therefore, can progress through a carefully designed sequence of units, starting with simple, generally applicable fundamentals and ending with more complex and specialized subjects.

Basic Group. Chapter 1 provides a historical perspective and some fundamental issues in hardware, software, and applications. Chapters 2 and 3 describe, respectively, the use and the implementation of SRGP, a simple 2D integer graphics package. Chapter 4 introduces graphics hardware, including some hints about how to use hardware in implementing the operations described in the preceding chapters. The next two chapters, 5 and 6, introduce the ideas of transformations in the plane and 3-space, representations by matrices, the use of homogeneous coordinates to unify linear and affine transformations, and the description of 3D views, including the transformations from arbitrary view volumes to canonical view volumes. Finally, Chapter 7 introduces SPHIGS, a 3D floating-point hierarchical graphics package that is a simplified version of the PHIGS standard, and describes its use in some basic modeling operations. Chapter 7 also discusses the advantages and disadvantages of the hierarchy available in PHIGS and the structure of applications that use this graphics package.

User Interface Group. Chapters 8-10 describe the current technology of interaction devices and then address the higher-level issues in user-interface design. Various popular user-interface paradigms are described and critiqued. In the final chapter user-interface software, such as window managers, interaction technique-libraries, and user-interface management systems, is addressed.

Model Definition Group. The first two modeling chapters, 11 and 12, describe the current technologies used in geometric modeling: the representation of curves and surfaces by parametric functions, especially cubic splines, and the representation of solids by various techniques, including boundary representations and CSG models. Chapter 13 introduces the human color-vision system, various color-description systems, and conversion from one to another. This chapter also briefly addresses rules for the effective use of color.

Image Synthesis Group. Chapter 14, the first in a four-chapter sequence, describes the quest for realism from the earliest vector drawings to state-of-the-art shaded graphics. The artifacts caused by aliasing are of crucial concern in raster graphics, and this chapter discusses their causes and cures in considerable detail by introducing the Fourier

Samsung/Google, EX1027, p. 11

transform and convolution. Chapter 15 describes a variety of strategies for visible-surface determination in enough detail to allow the reader to implement some of the most important ones. Illumination and shading algorithms are covered in detail in Chapter 16. The early part of this chapter discusses algorithms most commonly found in current hardware, while the remainder treats texture, shadows, transparency, reflections, physically based illumination models, ray tracing, and radiosity methods. The last chapter in this group, Chapter 17, describes both image manipulations, such as scaling, shearing, and rotating pixmaps, and image storage techniques, including various image-compression schemes.

Advanced Techniques Group. The last four chapters give an overview of the current state of the art (a moving target, of course). Chapter 18 describes advanced graphics hardware used in high-end commercial and research machines; this chapter was contributed by Steven Molnar and Henry Fuchs, authorities on high-performance graphics architectures. Chapter 19 describes the complex raster algorithms used for such tasks as scan-converting arbitary conics, generating antialiased text, and implementing page-description languages, such as PostScript. The final two chapters survey some of the most important techniques in the fields of high-level modeling and computer animation.

The first two groups cover only elementary material and thus can be used for a basic course at the undergraduate level. A follow-on course can then use the more advanced chapters. Alternatively, instructors can assemble customized courses by picking chapters out of the various groups.

For example, a course designed to introduce students to primarily 2D graphics would include Chapters 1 and 2, simple scan conversion and clipping from Chapter 3, a technology overview with emphasis on raster architectures and interaction devices from Chapter 4, homogeneous mathematics from Chapter 5, and 3D viewing only from a "how to use it" point of view from Sections 6.1 to 6.3. The User Interface Group, Chapters 8-10, would be followed by selected introductory sections and simple algorithms from the Image Synthesis Group, Chapters 14, 15, and 16.

A one-course general overview of graphics would include Chapters 1 and 2, basic algorithms from Chapter 3, raster architectures and interaction devices from Chapter 4, Chapter 5, and most of Chapters 6 and 7 on viewing and SPHIGS. The second half of the course would include sections on modeling from Chapters 11 and 13, on image synthesis from Chapters 14, 15, and 16, and on advanced modeling from Chapter 20 to give breadth of coverage in these slightly more advanced areas.

A course emphasizing 3D modeling and rendering would start with Chapter 3 sections on scan converting, clipping of lines and polygons, and introducing antialiasing. The course would then progress to Chapters 5 and 6 on the basic mathematics of transformations and viewing, Chapter 13 on color, and then cover the key Chapters 14, 15, and 16 in the Image Synthesis Group. Coverage would be rounded off by selections in surface and solid modeling, Chapter 20 on advanced modeling, and Chapter 21 on animation from the Advanced Techniques Group.

Graphics Packages. The SRGP and SPHIGS graphics packages, designed by David Sklar, coauthor of the two chapters on these packages, are available from the publisher for

Samsung/Google, EX1027, p. 12

the IBM PC (ISBN 0-201-54700-7), the Macintosh (ISBN 0-201-54701-5), and UNIX workstations running X11, as are many of the algorithms for scan conversion, clipping, and viewing (see page 1175).

Acknowledgments. This book could not have been produced without the dedicated work and the indulgence of many friends and colleagues. We acknowledge here our debt to those who have contributed significantly to one or more chapters; many others have helped by commenting on individual chapters, and we are grateful to them as well. We regret any inadvertent omissions. Katrina Avery and Lyn Dupré did a superb job of editing. Additional valuable editing on multiple versions of multiple chapters was provided by Debbie van Dam, Melissa Gold, and Clare Campbell. We are especially grateful to our production supervisor, Bette Aaronson, our art director, Joe Vetere, and our editor, Keith Wollman, not only for their great help in producing the book, but also for their patience and good humor under admittedly adverse circumstances—if we ever made a promised deadline during these frantic five years, we can't remember it!

Computer graphics has become too complex for even a team of four main authors and three guest authors to be expert in all areas. We relied on colleagues and students to amplify our knowledge, catch our mistakes and provide constructive criticism of form and content. We take full responsibility for any remaining sins of omission and commission. Detailed technical readings on one or more chapters were provided by John Airey, Kurt Akeley, Tom Banchoff, Brian Barsky, David Bates, Cliff Beshers, Gary Bishop, Peter Bono, Marvin Bunker, Bill Buxton, Edward Chang, Norman Chin, Michael F. Cohen, William Cowan, John Dennis, Tom Dewald, Scott Draves, Steve Drucker, Tom Duff, Richard Economy, David Ellsworth, Nick England, Jerry Farrell, Robin Forrest, Alain Fournier, Alan Freiden, Christina Gibbs, Melissa Gold, Mark Green, Cathleen Greenberg, Margaret Hagen, Griff Hamlin, Pat Hanrahan, John Heidema, Rob Jacob, Abid Kamran, Mike Kappel, Henry Kaufman, Karen Kendler, David Kurlander, David Laidlaw, Keith Lantz, Hsien-Che Lee, Aaron Marcus, Nelson Max, Deborah Mayhew, Barbara Meier, Gary Meyer, Jim Michener, Jakob Nielsen, Mark Nodine, Randy Pausch, Ari Requicha, David Rosenthal, David Salesin, Hanan Samet, James Sanford, James Sargent, Robin Schaufler, Robert Scheifler, John Schnizlein, Michael Shantzis, Ben Shneiderman, Ken Shoemake, Judith Schrier, John Sibert, Dave Simons, Jonathan Steinhart, Maureen Stone, Paul Strauss, Seth Tager, Peter Tanner, Brice Tebbs, Ben Trumbore, Yi Tso, Greg Turk, Jeff Vroom, Colin Ware, Gary Watkins, Chuck Weger, Kevin Weiler, Turner Whitted, George Wolberg, and Larry Wolff.

Several colleagues, including Jack Bresenham, Brian Barsky, Jerry Van Aken, Dilip Da Silva (who suggested the uniform midpoint treatment of Chapter 3) and Don Hatfield, not only read chapters closely but also provided detailed suggestions on algorithms.

Welcome word-processing relief was provided by Katrina Avery, Barbara Britten, Clare Campbell, Tina Cantor, Joyce Cavatoni, Louisa Hogan, Jenni Rodda, and Debbie van Dam. Drawings for Chapters 1–3 were ably created by Dan Robbins, Scott Snibbe, Tina Cantor, and Clare Campbell. Figure and image sequences created for this book were provided by Beth Cobb, David Kurlander, Allen Paeth, and George Wolberg (with assistance from Peter Karp). Plates II.21–37, showing a progression of rendering techniques, were designed and rendered at Pixar by Thomas Williams and H.B. Siegel, under the direction of M.W. Mantle, using Pixar's PhotoRealistic RenderMan software. Thanks to Industrial Light &

Samsung/Google, EX1027, p. 13

Magic for the use of their laser scanner to create Plates II.24–37, and to Norman Chin for computing vertex normals for Color Plates II.30–32. L. Lu and Carles Castellsagué wrote programs to make figures.

Jeff Vogel implemented the algorithms of Chapter 3, and he and Atul Butte verified the code in Chapters 2 and 7. David Sklar wrote the Mac and X11 implementations of SRGP and SPHIGS with help from Ron Balsys, Scott Boyajian, Atul Butte, Alex Contovounesios, and Scott Draves. Randy Pausch and his students ported the packages to the PC environment.

We have installed an automated electronic mail server to allow our readers to obtain machine-readable copies of many of the algorithms, suggest exercises, report errors in the text and in SRGP/SPHIGS, and obtain errata lists for the text and software. Send email to "graphtext @ cs.brown.edu" with a Subject line of "Help" to receive the current list of available services. (See page 1175 for information on how to order SRGP and SPHIGS.)

Preface to the C Edition

This is the C-language version of a book originally written with examples in Pascal. It includes all changes through the ninth printing of the Pascal second edition, as well as minor modifications to several algorithms, and all its Pascal code has been rewritten in ANSI C. The interfaces to the SRGP and SPHIGS graphics packages are now defined in C, rather than Pascal, and correspond to the new C implementations of these packages. (See page 1175 for information on obtaining the software.)

We wish to thank Norman Chin for converting the Pascal code of the second edition to C, proofreading it, and formatting it using the typographic conventions of the original. Thanks to Matt Ayers for careful proofing of Chapters 2, 3, and 7, and for useful suggestions about conversion problems.

Washington, D.C. J.D.F.
Providence, R.I. A.v.D.
New York, N.Y. S.K.F.
Providence, R.I. J.F.H.

Samsung/Google, EX1027, p. 14

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

Samsung/Google, EX1027, p. 15

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

Samsung/Google, EX1027, p. 16

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

Samsung/Google, EX1027, p. 17

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

Samsung/Google, EX1027, p. 18

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

Samsung/Google, EX1027, p. 19

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

Samsung/Google, EX1027, p. 20

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**

Samsung/Google, EX1027, p. 21

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

Samsung/Google, EX1027, p. 22

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.

Samsung/Google, EX1027, p. 23

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

Samsung/Google, EX1027, p. 24

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

Samsung/Google, EX1027, p. 25

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

Samsung/Google, EX1027, p. 26

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

Samsung/Google, EX1027, p. 27

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

Samsung/Google, EX1027, p. 28

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], 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,

Samsung/Google, EX1027, p. 29

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 1.1 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

Samsung/Google, EX1027, p. 30

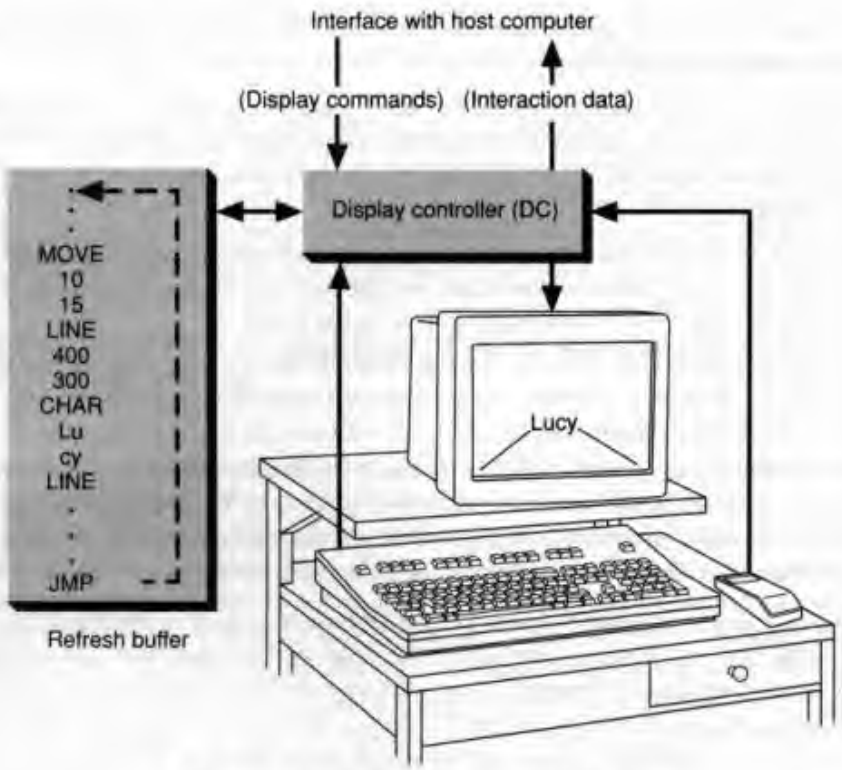


Fig. 1.1 Architecture of a vector display.

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

Samsung/Google, EX1027, p. 31

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

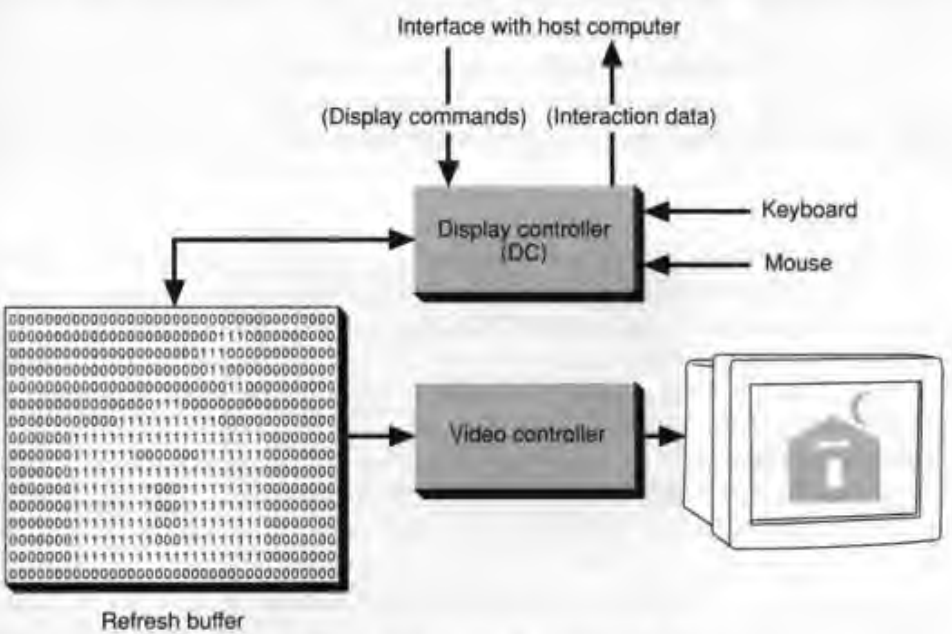


Fig. 1.2 Architecture of a raster display.

Samsung/Google, EX1027, p. 32

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,

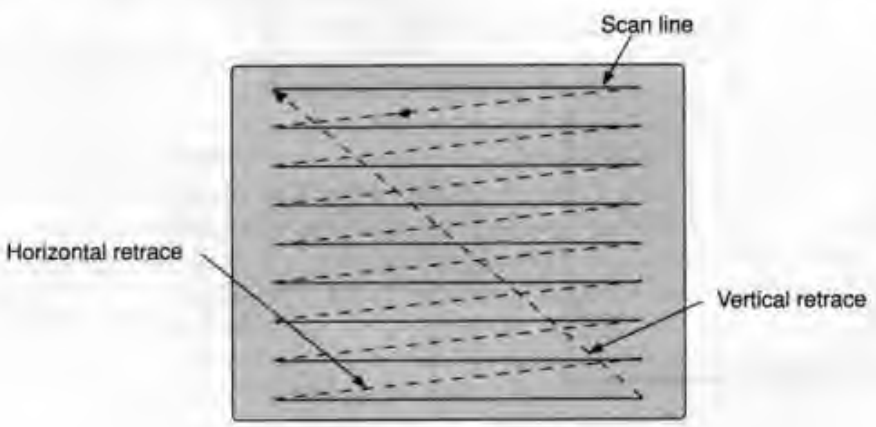


Fig. 1.3 Raster scan.

Samsung/Google, EX1027, p. 33

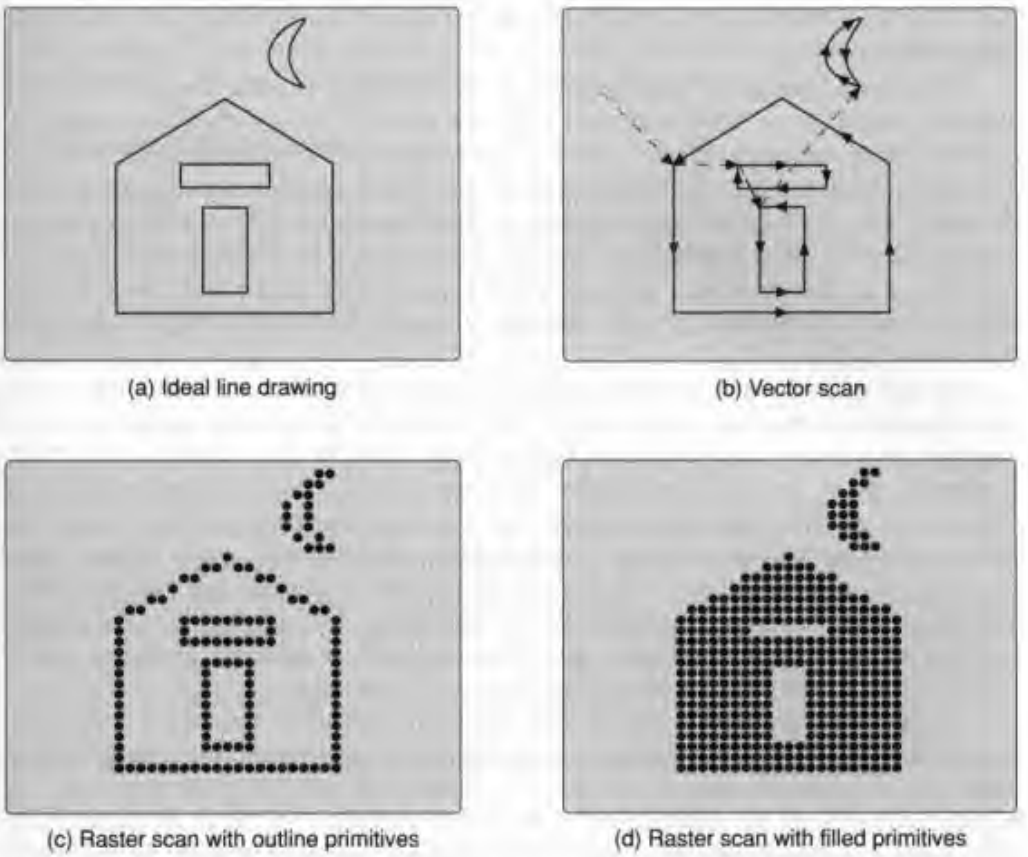


Fig. 1.4 Random scan versus raster scan. We symbolize the screen as a rounded rectangle filled with a light gray shade that denotes the white background; the image is drawn in black on this background.

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).

Samsung/Google, EX1027, p. 34

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

Samsung/Google, EX1027, p. 35

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*

Samsung/Google, EX1027, p. 36

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).

Samsung/Google, EX1027, p. 37

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.

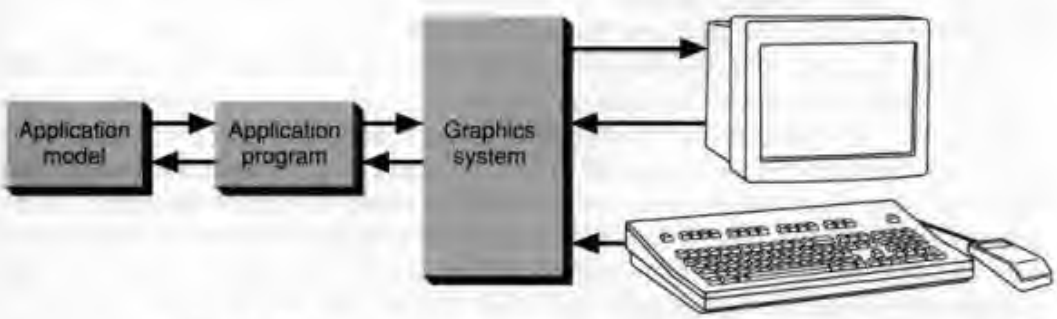


Fig. 1.5 Conceptual framework for interactive graphics.

Samsung/Google, EX1027, p. 38

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

Samsung/Google, EX1027, p. 39

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

Samsung/Google, EX1027, p. 40

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

Samsung/Google, EX1027, p. 41

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

Samsung/Google, EX1027, p. 42

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

Samsung/Google, EX1027, p. 43

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.

Samsung/Google, EX1027, p. 44

2
Programming in the Simple Raster Graphics Package (SRGP)

Andries van Dam
and David F. Sklar

In Chapter 1, we saw that vector and raster displays are two substantially different hardware technologies for creating images on the screen. Raster displays are now the dominant hardware technology, because they support several features that are essential to the majority of modern applications. First, raster displays can fill areas with a uniform color or a repeated pattern in two or more colors; vector displays can, at best, only simulate filled areas with closely spaced sequences of parallel vectors. Second, raster displays store images in a way that allows manipulation at a fine level: individual pixels can be read or written, and arbitrary portions of the image can be copied or moved.

The first graphics package we discuss, SRGP (Simple Raster Graphics Package), is a device-independent graphics package that exploits raster capabilities. SRGP's repertoire of primitives (lines, rectangles, circles and ellipses, and text strings) is similar to that of the popular Macintosh QuickDraw raster package and that of the Xlib package of the X Window System. Its interaction-handling features, on the other hand, are a subset of those of SPHIGS, the higher-level graphics package for displaying 3D primitives (covered in Chapter 7). SPHIGS (Simple PHIGS) is a simplified dialect of the standard PHIGS graphics package (Programmer's Hierarchical Interactive Graphics System) designed for both raster and vector hardware. Although SRGP and SPHIGS were written specifically for this text, they are also very much in the spirit of mainstream graphics packages, and most of what you will learn here is immediately applicable to commercial packages. In this book, we introduce both packages; for a more complete description, you should consult the reference manuals distributed with the software packages.

We start our discussion of SRGP by examining the operations that applications perform in order to draw on the screen: the specification of primitives and of the attributes that affect

Samsung/Google, EX1027, p. 45

their image. (Since graphics printers display information essentially as raster displays do, we need not concern ourselves with them until we look more closely at hardware in Chapter 4.) Next we learn how to make applications interactive using SRGP's input procedures. Then we cover the utility of pixel manipulation, available only in raster displays. We conclude by discussing some limitations of integer raster graphics packages such as SRGP.

Although our discussion of SRGP assumes that it controls the entire screen, the package has been designed to run in window environments (see Chapter 10), in which case it controls the interior of a window as though it were a virtual screen. The application programmer therefore does not need to be concerned about the details of running under control of a window manager.

2.1 DRAWING WITH SRGP

2.1.1 Specification of Graphics Primitives

Drawing in integer raster graphics packages such as SRGP is like plotting graphs on graph paper with a very fine grid. The grid varies from 80 to 120 points per inch on conventional displays to 300 or more on high-resolution displays. The higher the resolution, the better the appearance of fine detail. Figure 2.1 shows a display screen (or the surface of a printer's paper or film) ruled in SRGP's integer Cartesian coordinate system. Note that pixels in SRGP lie at the intersection of grid lines.

The origin (0, 0) is at the bottom left of the screen; positive x increases toward the right and positive y increases toward the top. The pixel at the upper-right corner is (width–1, height–1), where width and height are the device-dependent dimensions of the screen.

On graph paper, we can draw a continuous line between two points located anywhere on the paper; on raster displays, however, we can draw lines only between grid points, and the line must be approximated by intensifying the grid-point pixels lying on it or nearest to it. Similarly, solid figures such as filled polygons or circles are created by intensifying the pixels in their interiors and on their boundaries. Since specifying each pixel of a line or closed figure would be far too onerous, graphics packages let the programmer specify primitives such as lines and polygons via their vertices; the package then fills in the details using scan-conversion algorithms, discussed in Chapter 3.

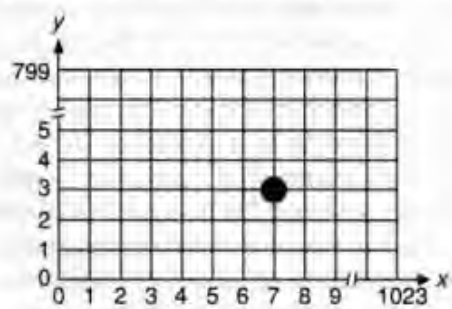


Fig. 2.1 Cartesian coordinate system of a screen 1024 pixels wide by 800 pixels high. Pixel (7, 3) is shown.

Samsung/Google, EX1027, p. 46

SRGP supports a basic collection of primitives: lines, polygons, circles and ellipses, and text.[1] To specify a primitive, the application sends the coordinates defining the primitive's shape to the appropriate SRGP primitive-generator procedure. It is legal for a specified point to lie outside the screen's bounded rectangular area; of course, only those portions of a primitive that lie inside the screen bounds will be visible.

Lines and polylines. The following SRGP procedure draws a line from (*x1*, *y1*) to (*x2*, *y2*):

```
void SRGP_lineCoord (int x1, int y1, int x2, int y2);
```

Thus, to plot a line from (0, 0) to (100, 300), we simply call

```
SRGP_lineCoord (0, 0, 100, 300);
```

Because it is often more natural to think in terms of endpoints rather than of individual *x* and *y* coordinates, SRGP provides an alternate line-drawing procedure:

```
void SRGP_line (point pt1, point pt2);
```

Here "point" is a defined type, a record of two integers holding the point's *x* and *y* values:

```
typedef struct {
    int x, y;
} point;
```

A sequence of lines connecting successive vertices is called a *polyline*. Although polylines can be created by repeated calls to the line-drawing procedures, SRGP includes them as a special case. There are two polyline procedures, analogous to the coordinate and point forms of the line-drawing procedures. These take arrays as parameters:

```
void SRGP_polyLineCoord (int vertexCount, int *xArray, int *yArray);
void SRGP_polyLine (int vertexCount, point *vertices);
```

where "*xArray*," "*yArray*," and "*vertices*" are pointers to user-declared arrays—arrays of integers, integers, and points, respectively.

The first parameter in both of these polyline calls tells SRGP how many vertices to expect. In the first call, the second and third parameters are integer arrays of paired *x* and *y* values, and the polyline is drawn from vertex (*xArray*[0], *yArray*[0]), to vertex (*xArray*[1], *yArray*[1]), to vertex (*xArray*[2], *yArray*[2]), and so on. This form is convenient, for instance, when plotting data on a standard set of axes, where *xArray* is a predetermined set

---

[1] Specialized procedures that draw a single pixel or an array of pixels are described in the SRGP reference manual.

[2] We use C with the following typesetting conventions. C keywords and built-in types are in boldface and user-defined types are in normal face. Symbolic constants are in uppercase type, and variables are italicized. Comments are in braces, and pseudocode is italicized. For brevity, declarations of constants and variables are omitted when obvious.

Samsung/Google, EX1027, p. 47

of values of the independent variable and *yArray* is the set of data being computed or input by the user. As an example, let us plot the output of an economic analysis program that computes month-by-month trade figures and stores them in the 12-entry integer data array *balanceOfTrade*. We will start our plot at (200, 200). To be able to see the differences between successive points, we will graph them 10 pixels apart on the x axis. Thus, we will create an integer array, *months*, to represent the 12 months, and will set the entries to the desired x values, 200, 210, . . ., 310. Similarly, we must increment each value in the data array by 200 to put the 12 y coordinates in the right place. Then, the graph in Fig. 2.2 is plotted with the following code:

```
/* Plot the axes */
SRGP_lineCoord (175, 200, 320, 200);
SRGP_lineCoord (200, 140, 200, 280);

/* Plot the data */
SRGP_polyLineCoord (12, months, balanceOfTrade);
```

We can use the second polyline form to draw shapes by specifying pairs of x and y values together as points, passing an array of such points to SRGP. We create the bowtie in Fig. 2.3 by calling

```
SRGP_polyLine (7, bowtieArray);
```

The table in Fig. 2.3 shows how *bowtieArray* was defined.

Markers and polymarkers. It is often convenient to place *markers* (e.g., dots, asterisks, or circles) at the data points on graphs. SRGP therefore offers companions to the line and polyline procedures. The following procedures will create a marker symbol centered at (x, y):

```
void SRGP_markerCoord (int x, int y);
void SRGP_marker (point pt);
```

The marker's style and size can be changed as well, as explained in Section 2.1.2. To create

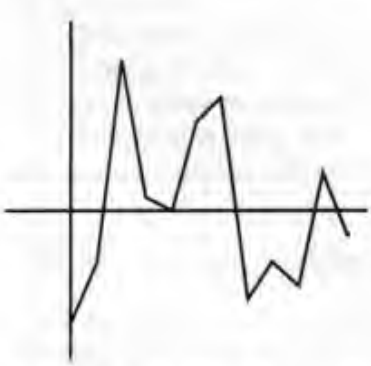
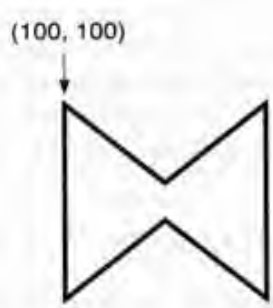


Fig. 2.2 Graphing a data array.



	x	y
0	100	100
1	100	60
2	120	76
3	140	60
4	140	100
5	120	84
6	100	100

bowtieArray

Fig. 2.3 Drawing a polyline.

Samsung/Google, EX1027, p. 48

a sequence of identical markers at a set of points, we call either of

```
void SRGP_polyMarkerCoord (int vertexCount, int *xArray, int *yArray);
void SRGP_polyMarker (int vertexCount, point *vertices);
```

Thus, the following additional call will add markers to the graph of Fig. 2.2 to produce Fig. 2.4.

```
SRGP_polyMarkerCoord (12, months, balanceOfTrade);
```

Polygons and rectangles. To draw an outline polygon, we can either specify a polyline that closes on itself by making the first and last vertices identical (as we did to draw the bowtie in Fig. 2.3), or we can use the following specialized SRGP call:

```
void SRGP_polygon (int vertexCount, point *vertices);
```

This call automatically closes the figure by drawing a line from the last vertex to the first. To draw the bowtie in Fig. 2.3 as a polygon, we use the following call, where *bowtieArray* is now an array of only six points:

```
SRGP_polygon (6, bowtieArray);
```

Any rectangle can be specified as a polygon having four vertices, but an upright rectangle (one whose edges are parallel to the screen's edges) can also be specified with the SRGP "rectangle" primitive using only two vertices (the lower-left and the upper-right corners).

```
void SRGP_rectangleCoord (int leftX, int bottomY, int rightX, int topY);
void SRGP_rectanglePt (point bottomLeft, point topRight);
void SRGP_rectangle (rectangle rect);
```

The "rectangle" record stores the bottom-left and top-right corners:

```
typedef struct {
    point bottomLeft, topRight;
} rectangle;
```

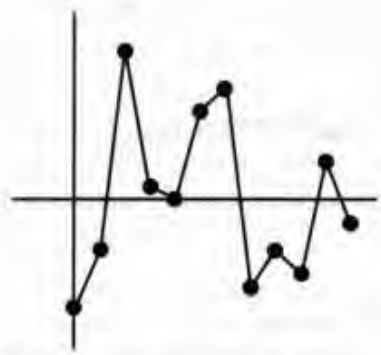


Fig. 2.4 Graphing the data array using markers.

Samsung/Google, EX1027, p. 49

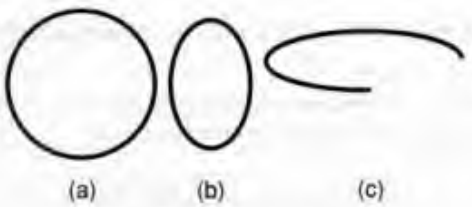


Fig. 2.5 Ellipse arcs.

Thus the following call draws an upright rectangle 101 pixels wide and 151 pixels high:

```
SRGP_rectangleCoord (50, 25, 150, 175);
```

SRGP provides the following utilities for creating rectangles and points from coordinate data.

```
point SRGP_defPoint (int x, int y);
rectangle SRGP_defRectangle (int leftX, int bottomY, int rightX, int topY);
```

Our example rectangle could thus have been drawn by

```
rect = SRGP_defRectangle (50, 25, 150, 175);
SRGP_rectangle (rect);
```

Circles and ellipses. Figure 2.5 shows circular and elliptical arcs drawn by SRGP. Since circles are a special case of ellipses, we use the term *ellipse arc* for all these forms, whether circular or elliptical, closed or partial arcs. SRGP can draw only standard ellipses, those whose major and minor axes are parallel to the coordinate axes.

Although there are many mathematically equivalent methods for specifying ellipse arcs, it is convenient for the programmer to specify arcs via the upright rectangles in which they are inscribed (see Fig. 2.6); these upright rectangles are called *bounding boxes* or *extents*.

The width and height of the extent determine the shape of the ellipse. Whether or not the arc is closed depends on a pair of angles that specify where the arc starts and ends. For convenience, each angle is measured in *rectangular degrees* that run counterclockwise, with 0° corresponding to the positive portion of the x axis, 90° to the positive portion of the y

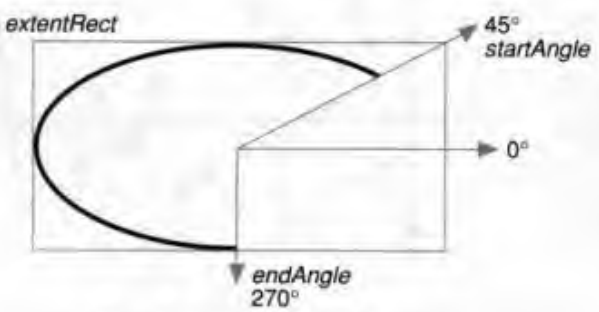


Fig. 2.6 Specifying ellipse arcs.

Samsung/Google, EX1027, p. 50

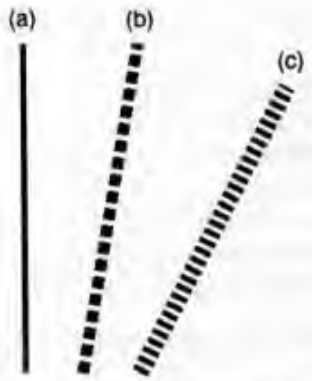


Fig. 2.7 Lines of various widths and styles.

axis, and 45° to the "diagonal" extending from the origin to the top-right corner of the rectangle. Clearly, only if the extent is a square are rectangular degrees equivalent to circular degrees.

The general ellipse procedure is

void SRGP_ellipseArc (rectangle *extentRect*, **double** *startAngle*, **double** *endAngle*);

2.1.2 Attributes

Line style and line width. The appearance of a primitive can be controlled by specification of its *attributes*.[3] The SRGP attributes that apply to lines, polylines, polygons, rectangles, and ellipse arcs are *line style*, *line width*, *color*, and *pen style*.

Attributes are set *modally*; that is, they are global state variables that retain their values until they are changed explicitly. Primitives are drawn with the attributes in effect at the time the primitives are specified; therefore, changing an attribute's value in no way affects previously created primitives—it affects only those that are specified after the change in attribute value. Modal attributes are convenient because they spare programmers from having to specify a long parameter list of attributes for each primitive, since there may be dozens of different attributes in a production system.

Line style and line width are set by calls to

void SRGP_setLineStyle (CONTINUOUS / DASHED / DOTTED/... lineStyle);[4]
void SRGP_setLineWidth (**int** *width*);

The width of a line is measured in screen units—that is, in pixels. Each attribute has a default: line style is CONTINUOUS, and width is 1. Figure 2.7 shows lines in a variety of widths and styles; the code that generated the figure is shown in Fig. 2.8.

---

[3]The descriptions here of SRGP's attributes often lack fine detail, particularly on interactions between different attributes. The detail is omitted because the exact effect of an attribute is a function of its implementation, and, for performance reasons, different implementations are used on different systems; for these details, consult the implementation-specific reference manuals.

[4]Here and in the following text, we use a shorthand notation. In SRGP, these symbolic constants are actually values of an enumerated data type "lineStyle."

Samsung/Google, EX1027, p. 51

```
SRGP_setLineWidth (5);
SRGP_lineCoord (55, 5, 55, 295);        /* Line a */

SRGP_setLineStyle (DASHED);
SRGP_setLineWidth (10);
SRGP_lineCoord (105, 5, 155, 295);      /* Line b */

SRGP_setLineWidth (15);
SRGP_setLineStyle (DOTTED);
SRGP_lineCoord (155, 5, 285, 255)       /* Line c */
```

Fig. 2.8 Code used to generate Fig. 2.7.

We can think of the line style as a bit mask used to write pixels selectively as the primitive is scan-converted by SRGP. A zero in the mask indicates that this pixel should not be written and thus preserves the original value of this pixel in the frame buffer. One can think of this pixel of the line as transparent, in that it lets the pixel "underneath" show through. CONTINUOUS thus corresponds to the string of all 1s, and DASHED to the string 1111001111001111 . . ., the dash being twice as long as the transparent interdash segments.

Each attribute has a default; for example, the default for line style is CONTINUOUS, that for line width is 1, and so on. In the early code examples, we did not set the line style for the first line we drew; thus, we made use of the line-style default. In practice, however, making assumptions about the current state of attributes is not safe, and in the code examples that follow we set attributes explicitly in each procedure, so as to make the procedures modular and thus to facilitate debugging and maintenance. In Section 2.1.4, we see that it is even safer for the programmer to save and restore attributes explicitly for each procedure.

Attributes that can be set for the marker primitive are

```
void SRGP_setMarkerSize (int markerSize);
void SRGP_setMarkerStyle (MARKER_CIRCLE / MARKER_SQUARE/. . . markerStyle);
```

Marker size specifies the length in pixels of the sides of the square extent of each marker. The complete set of marker styles is presented in the reference manual; the circle style is the default shown in Fig. 2.4.

Color. Each of the attributes presented so far affects only some of the SRGP primitives, but the integer-valued color attribute affects all primitives. Obviously, the color attribute's meaning is heavily dependent on the underlying hardware; the two color values found on every system are 0 and 1. On bilevel systems, these colors' appearances are easy to predict—color-1 pixels are black and color-0 pixels are white for black-on-white devices, green is 1 and black is 0 for green-on-black devices, and so on.

The integer color attribute does not specify a color directly; rather, it is an index into SRGP's *color table*, each entry of which defines a color or gray-scale value in a manner that the SRGP programmer does not need to know about. There are 2^d entries in the color table, where d is the *depth* (number of bits stored for each pixel) of the frame buffer. On bilevel implementations, the color table is hardwired; on most color implementations, however,

Samsung/Google, EX1027, p. 52

SRGP allows the application to modify the table. Some of the many uses for the indirectness provided by color tables are explored in Chapters 4, 17, and 21.

There are two methods that applications can use to specify colors. An application for which machine independence is important should use the integers 0 and 1 directly; it will then run on all bilevel and color displays. If the application assumes color support or is written for a particular display device, then it can use the implementation-dependent *color names* supported by SRGP. These names are symbolic constants that show where certain standard colors have been placed within the default color table for that display device. For instance, a black-on-white implementation provides the two color names COLOR_BLACK (1) and COLOR_WHITE (0); we use these two values in the sample code fragments in this chapter. Note that color names are not useful to applications that modify the color table.

We select a color by calling

void SRGP_setColor (**int** *colorIndex*);

2.1.3 Filled Primitives and Their Attributes

Primitives that enclose areas (the so-called *area-defining* primitives) can be drawn in two ways: *outlined* or *filled*. The procedures described in the previous section generate the former style: closed outlines with unfilled interiors. SRGP's filled versions of area-defining primitives draw the interior pixels with no outline. Figure 2.9 shows SRGP's repertoire of filled primitives, including the filled ellipse arc, or *pie slice*.

Note that SRGP does not draw a contrasting outline, such as a 1-pixel-thick solid boundary, around the interior; applications wanting such an outline must draw it explicitly. There is also a subtle issue of whether pixels on the border of an area-defining primitive should actually be drawn or whether only pixels that lie strictly in the interior should. This problem is discussed in detail in Sections 3.5 and 3.6.

To generate a filled polygon, we use SRGP_fillPolygon or SRGP_fillPolygonCoord, with the same parameter lists used in the unfilled versions of these calls. We define the other area-filling primitives in the same way, by prefixing "fill" to their names. Since polygons

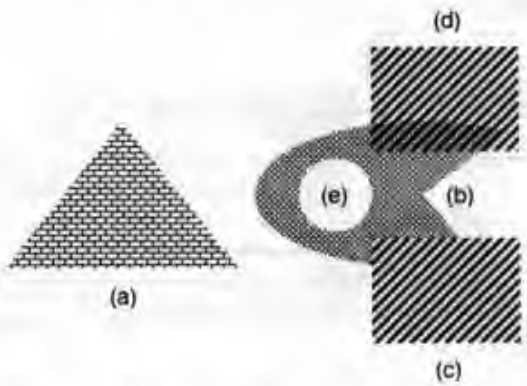


Fig. 2.9 Filled primitives. (a-c) Bitmap pattern opaque. (d) Bitmap pattern transparent. (e) Solid.

Samsung/Google, EX1027, p. 53

may be concave or even self-intersecting, we need a rule for specifying what regions are interior and thus should be filled, and what regions are exterior. SRGP polygons follow the *odd-parity* rule. To determine whether a region lies inside or outside a given polygon, choose as a test point any point inside the particular region. Next, choose a ray that starts at the test point and extends infinitely in any direction, and that does not pass through any vertices. If this ray intersects the polygon outline an odd number of times, the region is considered to be interior (see Fig. 2.10).

SRGP does not actually perform this test for each pixel while drawing; rather, it uses the optimized polygon scan-conversion techniques described in Chapter 3, in which the odd-parity rule is efficiently applied to an entire row of adjacent pixels that lie either inside or outside. Also, the odd-parity ray-intersection test is used in a process called *pick correlation* to determine the object a user is selecting with the cursor, as described in Chapter 7.

Fill style and fill pattern for areas. The fill-style attribute can be used to control the appearance of a filled primitive's interior in four different ways, using

```
void SRGP_setFillStyle (
    SOLID / BITMAP_PATTERN_OPAQUE / BITMAP_PATTERN_TRANSPARENT /
    PIXMAP_PATTERN drawStyle);
```

The first option, SOLID, produces a primitive uniformly filled with the current value of the color attribute (Fig. 2.9e, with color set to COLOR_WHITE). The second two options, BITMAP_PATTERN_OPAQUE and BITMAP_PATTERN_TRANSPARENT, fill primitives with a regular, nonsolid pattern, the former rewriting all pixels underneath in either the current color, or another color (Fig. 2.9c), the latter rewriting some pixels underneath the primitive in the current color, but letting others show through (Fig. 2.9d). The last option, PIXMAP_PATTERN, writes patterns containing an arbitrary number of colors, always in opaque mode.

Bitmap fill patterns are bitmap arrays of 1s and 0s chosen from a table of available

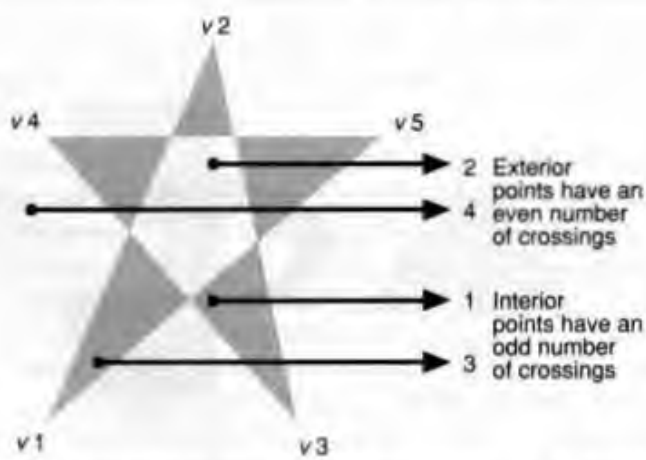


Fig. 2.10 Odd-parity rule for determining interior of a polygon.

Samsung/Google, EX1027, p. 54

patterns by specifying

void SRGP_setFillBitmapPattern (**int** *patternIndex*);

Each entry in the pattern table stores a unique pattern; the ones provided with SRGP, shown in the reference manual, include gray-scale tones (ranging from nearly black to nearly white) and various regular and random patterns. In transparent mode, these patterns are generated as follows. Consider any pattern in the pattern table as a small bitmap—say, 8 by 8—to be repeated as needed (*tiled*) to fill the primitive. On a bilevel system, the current color (in effect, the *foreground* color) is written where there are 1s in the pattern; where there are 0s—the "holes"—the corresponding pixels of the original image are not written, and thus "show through" the partially transparent primitive written on top. Thus, the bitmap pattern acts as a "memory write-enable mask" for patterns in transparent mode, much as the line-style bit mask did for lines and outline primitives.

In the more commonly used BITMAP_PATTERN_OPAQUE mode, the 1s are written in the current color, but the 0s are written in another color, the *background color*, previously set by

void SRGP_setBackgroundColor (**int** *colorIndex*);

On bilevel displays, each bitmap pattern in OPAQUE mode can generate only two distinctive fill patterns. For example, a bitmap pattern of mostly 1s can be used on a black-and-white display to generate a dark-gray fill pattern if the current color is set to black (and the background to white), and a light-gray fill pattern if the current color is set to white (and the background to black). On a color display, any combination of a foreground and a background color may be used for a variety of two-tone effects. A typical application on a bilevel display always sets the background color whenever it sets the foreground color, since opaque bitmap patterns are not visible if the two are equal; an application could create a SetColor procedure to set the background color automatically to contrast with the foreground whenever the foreground color is set explicitly.

Figure 2.9 was created by the code fragment shown in Fig. 2.11. The advantage of

```
SRGP_setFillStyle (BITMAP_PATTERN_OPAQUE);
SRGP_setFillBitmapPattern (BRICK_BIT_PATTERN);          /* Brick pattern */
SRGP_fillPolygon (3, triangleCoords);                   /* a */

SRGP_setFillBitmapPattern (MEDIUM_GRAY_BIT_PATTERN);    /* 50 percent gray */
SRGP_fillEllipseArc (ellipseArcRect, 60.0, 290.0);      /* b */

SRGP_setFillBitmapPattern (DIAGONAL_BIT_PATTERN);
SRGP_fillRectangle (opaqueFilledRect);                  /* c */

SRGP_setFillStyle (BITMAP_PATTERN_TRANSPARENT);
SRGP_fillRectangle (transparentFilledRect);             /* d */

SRGP_setFillStyle (SOLID);
SRGP_setColor (COLOR_WHITE);
SRGP_fillEllipse (circleRect);                          /* e */
```

Fig. 2.11 Code used to generate Fig. 2.9.

Samsung/Google, EX1027, p. 55

having two-tone bitmap patterns is that the colors are not specified explicitly, but rather are determined by the color attributes in effect, and thus can be generated in any color combination. The disadvantage, and the reason that SRGP also supports pixmap patterns, is that only two colors can be generated. Often, we would like to fill an area of a display with multiple colors, in an explicitly specified pattern. In the same way that a bitmap pattern is a small bitmap used to tile the primitive, a small pixmap can be used to tile the primitive, where the pixmap is a pattern array of color-table indices. Since each pixel is explicitly set in the pixmap, there is no concept of holes, and therefore there is no distinction between transparent and opaque filling modes. To fill an area with a color pattern, we select a fill style of PIXMAP_PATTERN and use the corresponding pixmap pattern-selection procedure:

void SRGP_setFillPixmapPattern (**int** *patternIndex*);

Since both bitmap and pixmap patterns generate pixels with color values that are indices into the current color table, the appearance of filled primitives changes if the programmer modifies the color-table entries. The SRGP reference manual discusses how to change or add to both the bitmap and pixmap pattern tables. Also, although SRGP provides default entries in the bitmap pattern table, it does not give a default pixmap pattern table, since there is an indefinite number of color pixmap patterns that might be useful.

Pen pattern for outlines. The advantages of patterning are not restricted to the use of this technique in area-defining primitives; patterning can also be used to affect the appearance of lines and outline primitives, via the *pen-style* attribute. Using the line-width, line-style, and pen-style attributes, it is possible, for example, to create a 5-pixel-thick, dot-dashed ellipse whose thick dashes are patterned. Examples of solid and dashed thick lines with various patterns in transparent and opaque mode and their interactions with previously drawn primitives are shown in Fig. 2.12; the code that generated the image is in Fig. 2.13. The use of a pen pattern for extremely narrow lines (1 or 2 pixels wide) is not recommended, because the pattern is not discernible in such cases.

The interaction between line style and pen style is simple: 0s in the line-style mask

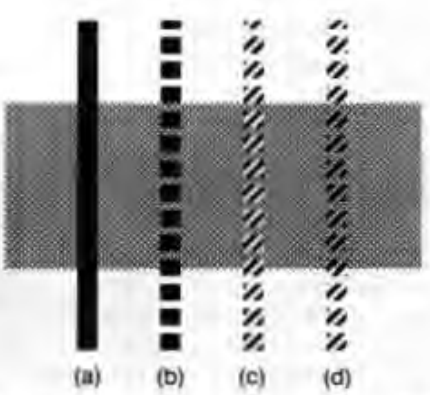


Fig. 2.12 Interaction between pen style and line style. (a) Continuous solid. (b) Dashed solid. (c) Dashed bitmap pattern opaque. (d) Dashed bitmap pattern transparent.

Samsung/Google, EX1027, p. 56

```
/* We show only the drawing of the lines, not the background rectangle. */
/* We draw the lines in order from left to right */

SRGP_setLineWidth (15);      /* Thick lines show the interaction better. */

SRGP_setLineStyle (CONTINUOUS);
SRGP_setPenStyle (SOLID);
SRGP_line (pta1, pta2);      /* a: Solid, continuous */

SRGP_setLineStyle (DASHED);
SRGP_line (ptb1, ptb2);      /* b: Solid, dashed */

SRGP_setPenBitmapPattern (DIAGONAL_BIT_PATTERN);
SRGP_setPenStyle (BITMAP_PATTERN_OPAQUE);
SRGP_line (ptc1, ptc2);      /* c: Dashed, bitmap pattern opaque */

SRGP_setPenStyle (BITMAP_PATTERN_TRANSPARENT);
SRGP_line (ptd1, ptd2);      /* d: Dashed, bitmap pattern transparent */
```

Fig. 2.13 Code used to generate Fig. 2.12.

fully protect the pixels on which they fall, so the pen style influences only those pixels for which the line-style mask is 1.

Pen style is selected with the same four options and the same patterns as fill style. The same bitmap and pixmap pattern tables are also used, but separate indices are maintained so that resetting a pen style's pattern index will not affect the fill style's pattern index.

```
void SRGP_setPenStyle (SOLID / BITMAP_PATTERN_OPAQUE/... drawStyle);
void SRGP_setPenBitmapPattern (int patternIndex);
void SRGP_setPenPixmapPattern (int patternIndex);
```

Application screen background. We have defined "background color" as the color of the 0 bits in bitmap patterns used in opaque mode, but the term *background* is used in another, unrelated way. Typically, the user expects the screen to display primitives on some uniform *application screen background pattern* that covers an opaque window or the entire screen. The application screen background pattern is often solid color 0, since SRGP initializes the screen to that color upon initialization. However, the background pattern is sometimes nonsolid, or solid of some other color; in these cases, the application is responsible for setting up the application screen background by drawing a full-screen rectangle of the desired pattern, before drawing any other primitives.

A common technique to "erase" primitives is to redraw them in the application screen background pattern, rather than redrawing the entire image each time a primitive is deleted. However, this "quick and dirty" updating technique yields a damaged image when the erased primitive overlaps with other primitives. For example, assume that the screen background pattern in Fig. 2.9 is solid white and that we erase the rectangle marked (c) by redrawing it using solid COLOR_WHITE. This would leave a white gap in the filled ellipse arc (b) underneath. "Damage repair" involves going back to the application database and respecifying primitives (see Exercise 2.9).

Samsung/Google, EX1027, p. 57

2.1.4 Saving and Restoring Attributes

As you can see, SRGP supports a variety of attributes for its various primitives. Individual attributes can be saved for later restoration; this feature is especially useful in designing application procedures that perform their functions without side effects—that is, without affecting the global attribute state. For each attribute-setting SRGP procedure, there is a corresponding inquiry procedure that can be used to determine the current value; for example,

```
lineStyle SRGP_inquireLineStyle (void);
```

For convenience, SRGP allows the inquiry and restoration of the entire set of attributes—called the *attribute group*—via

```
void SRGP_inquireAttributes (attributeGroup *group);
void SRGP_setAttributes (attributeGroup *group);
```

In the current implementation of SRGP, unlike in previous versions, the application program can access the internal fields of the attributeGroup structure. Directly modifying these fields, however, is a bit tricky and the programmer does so at her own risk.

2.1.5 Text

Specifying and implementing text drawing is always complex in a graphics package, because of the large number of options and attributes text can have. Among these are the style or *font* of the characters (Times Roman, Helvetica, Clarinda, etc.), their appearance ("Roman," **bold**, *italic*, underlined, etc.), their size (typically measured in *points*[3]) and widths, the intercharacter spacing, the spacing between consecutive lines, the angle at which characters are drawn (horizontal, vertical, or at a specified angle), and so on.

The most rudimentary facility, typically found in simple hardware and software, is fixed-width, monospace character spacing, in which all characters occupy the same width, and the spacing between them is constant. At the other end of the spectrum, proportional spacing varies both the width of characters and the spacing between them to make the text as legible and aesthetically pleasing as possible. Books, magazines, and newspapers all use proportional spacing, as do most raster graphics displays and laser printers. SRGP provides in-between functionality: Text is horizontally aligned, character widths vary, but space between characters is constant. With this simple form of proportional spacing, the application can annotate graphics diagrams, interact with the user via textual menus and fill-in forms, and even implement simple word processors. Text-intensive applications, however, such as desktop-publishing programs for high-quality documents, need specialized packages that offer more control over text specification and attributes than does SRGP. PostScript [ADOB87] offers many such advanced features and has become an industry standard for describing text and other primitives with a large variety of options and attributes.

---

[3]A point is a unit commonly used in the publishing industry; it is equal to approximately 1/72 inch.

Samsung/Google, EX1027, p. 58

Text is generated by a call to

void SRGP_text (point *origin*, **char** **text*);

The location of a text primitive is controlled by specification of its *origin*, also known as its *anchor point*. The *x* coordinate of the origin marks the left edge of the first character, and the y coordinate specifies where the baseline of the string should appear. (The *baseline* is the hypothetical line on which characters rest, as shown in the textual menu button of Fig. 2.14. Some characters, such as "y" and "q," have a tail, called the *descender*, that goes below the baseline.)

A text primitive's appearance is determined by only two attributes, the current color and the font, which is an index into an implementation-dependent table of fonts in various sizes and styles:

void SRGP_setFont (**int** *fontIndex*);

Each character in a font is defined as a rectangular bitmap, and SRGP draws a character by filling a rectangle using the character's bitmap as a pattern, in bitmap-pattern-transparent mode. The 1s in the bitmap define the character's interior, and the 0s specify the surrounding space and gaps such as the hole in "o." (Some more sophisticated packages define characters in pixmaps, allowing a character's interior to be patterned.)

Formatting text. Because SRGP implementations offer a restricted repertoire of fonts and sizes, and because implementations on different hardware rarely offer equivalent repertoires, an application has limited control over the height and width of text strings. Since text-extent information is needed in order to produce well-balanced compositions (for instance, to center a text string within a rectangular frame), SRGP provides the following procedure for querying the extent of a given string using the current value of the font attribute:

void SRGP_inquireTextExtent (
    char **text*, **int** **width*, **int** **height*, **int** **descent*);

Although SRGP does not support bitmap opaque mode for writing characters, such a mode can be simulated easily. As an example, the procedure in Fig. 2.15 shows how extent

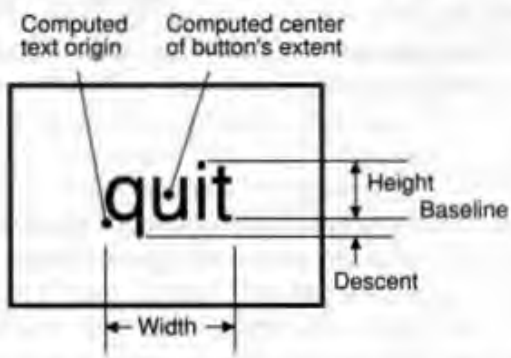


Fig. 2.14 Dimensions of text centered within a rectangular button and points computed from these dimensions for centering purposes.

Samsung/Google, EX1027, p. 59

```
void MakeQuitButton (rectangle buttonRect)
{
    point centerOfButton, textOrigin;
    int width, height, descent;

    SRGP_setFillStyle (SOLID);
    SRGP_setColor (COLOR_WHITE);
    SRGP_fillRectangle (buttonRect);
    SRGP_setColor (COLOR_BLACK);
    SRGP_setLineWidth (2);
    SRGP_Rectangle (buttonRect);

    SRGP_inquireTextExtent ("quit", &width, &height, &descent);

    centerOfButton.x = (buttonRect.bottomLeft.x + buttonRect.topRight.x) / 2;
    centerOfButton.y = (buttonRect.bottomLeft.y + buttonRect.topRight.y) / 2;

    textOrigin.x = centerOfButton.x - (width / 2);
    textOrigin.y = centerOfButton.y - (height / 2);

    SRGP_text (textOrigin,"quit");
} /* MakeQuitButton */
```

Fig. 2.15 Code used to create Fig. 2.14.

information and text-specific attributes can be used to produce black text, in the current font, centered within a white enclosing rectangle, as shown in Fig. 2.14. The procedure first creates the background button rectangle of the specified size, with a separate border, and then centers the text within it. Exercise 2.10 is a variation on this theme.

2.2 BASIC INTERACTION HANDLING

Now that we know how to draw basic shapes and text, the next step is to learn how to write interactive programs that communicate effectively with the user, using input devices such as the keyboard and the mouse. First, we look at general guidelines for making effective and pleasant-to-use interactive programs; then, we discuss the fundamental notion of logical (abstract) input devices. Finally, we look at SRGP's mechanisms for dealing with various aspects of interaction handling.

2.2.1 Human Factors

The designer of an interactive program must deal with many matters that do not arise in a noninteractive, batch program. These are the so-called *human factors* of a program, such as its interaction style (often called "look and feel") and its ease of learning and of use, and they are as important as its functional completeness and correctness. Techniques for user–computer interaction that exhibit good human factors are studied in more detail in Chapters 8 and 9. The guidelines discussed there include these:

Samsung/Google, EX1027, p. 60

- Provide *simple and consistent* interaction sequences.
- *Do not overload the user* with too many different options and styles.
- *Show the available options clearly* at every stage of the interaction.
- *Give appropriate feedback* to the user.
- Allow the users to *recover gracefully* from mistakes.

We attempt to follow these guidelines for good human factors in our sample programs. For example, we typically use menus to allow the user to indicate the next function he wants to execute by picking a text button in a menu of such buttons with a mouse. Also common are *palettes* (iconic menus) of basic geometric primitives, application-specific symbols, or fill patterns. Menus and palettes satisfy our first three guidelines in that their entries prompt the user with the list of available options and provide a single, consistent way of choosing among these options. Unavailable options may be either deleted temporarily or "grayed out" by being drawn in a low-intensity gray-scale pattern rather than a solid color (see Exercise 2.15).

Feedback occurs at every step of a menu operation to satisfy the fourth guideline: The application program will *highlight* the menu choice or object selection—for example, display it in inverse video or framed in a rectangle—to draw attention to it. The package itself may also provide an *echo* that gives an immediate response to the manipulation of an input device. For example, characters appear immediately at the position of the cursor as keyboard input is typed; as the mouse is moved on the table or desktop, a cursor echoes the corresponding location on the screen. Graphics packages offer a variety of cursor shapes that can be used by the application program to reflect the state of the program. In many display systems, the cursor shape can be varied dynamically as a function of the cursor's position on the screen. In many word-processing programs, for example, the cursor is shown as an arrow in menu areas and as a blinking vertical bar in text areas.

Graceful error recovery, our fifth guideline, is usually provided through *cancel* and *undo/redo* features. These require the application program to maintain a record of operations (see Chapter 9).

2.2.2 Logical Input Devices

Device types in SRGP. A major goal in designing graphics packages is device-independence, which enhances portability of applications. SRGP achieves this goal for graphics output by providing primitives specified in terms of an abstract integer coordinate system, thus shielding the application from the need to set the individual pixels in the frame buffer. To provide a level of abstraction for graphics input, SRGP supports a set of *logical input devices* that shield the application from the details of the physical input devices available. The two logical devices supported by SRGP are

- *Locator*, a device for specifying screen coordinates and the state of one or more associated buttons
- *Keyboard*, a device for specifying character string input

Samsung/Google, EX1027, p. 61

SRGP maps the logical devices onto the physical devices available (e.g., the locator could map to a mouse, joystick, tablet, or touch-sensitive screen). This mapping of logical to physical is familiar from conventional procedural languages and operating systems, where I/O devices such as terminals, disks, and tape drives are abstracted to logical data files to achieve both device-independence and simplicity of application programming.

Device handling in other packages. SRGP's input model is essentially a subset of the GKS and PHIGS input models. SRGP implementations support only one logical locator and one keyboard device, whereas GKS and PHIGS allow multiple devices of each type. Those packages also support additional device types: the *stroke* device (returning a polyline of cursor positions entered with the physical locator), the *choice* device (abstracting a function-key pad and returning a key identifier), the *valuator* (abstracting a slider or control dial and returning a floating-point number), and the *pick* device (abstracting a pointing device, such as a mouse or data tablet, with an associated button to signify a selection, and returning the identification of the logical entity picked). Other packages, such as QuickDraw and the X Window System, handle input devices in a more device-dependent way that gives the programmer finer control over an individual device's operation, at the cost of greater application-program complexity.

Chapter 8 presents the history of logical devices and elaborates further on their properties. Here, we briefly summarize modes of interacting with logical devices in general, and then examine SRGP's interaction procedures in more detail.

2.2.3 Sampling Versus Event-Driven Processing

There are two fundamental techniques for receiving information created by user interactions. In *sampling* (also called *polling*), the application program queries a logical input device's current value (called the *measure* of the device) and continues execution. The sampling is performed regardless of whether the device's measure has changed since the last sampling; indeed, only by continuous sampling of the device will changes in the device's state be known to the application. This mode is costly for interactive applications, because they would spend most of their CPU cycles in tight sampling loops waiting for measure changes.

An alternative to the CPU-intensive polling loop is the use of *interrupt-driven* interaction; in this technique, the application enables one or more devices for input and then continues normal execution until interrupted by some input *event* (a change in a device's state caused by user action); control then passes asynchronously to an interrupt procedure, which responds to the event. For each input device, an *event trigger* is defined; the event trigger is the user action that causes an event to occur. Typically, the trigger is a button push, such as a press of the mouse button ("mouse down") or a press of a keyboard key.

To free applications programmers from the tricky and difficult aspects of asynchronous transfer of control, many graphics packages, including GKS, PHIGS, and SRGP, offer *event-driven* interaction as a synchronous simulation of interrupt-driven interaction. In this technique, an application enables devices and then continues execution. In the background, the package monitors the devices and stores information about each event in an *event queue* (Fig. 2.16). The application, at its convenience, checks the event queue and processes the

Samsung/Google, EX1027, p. 62

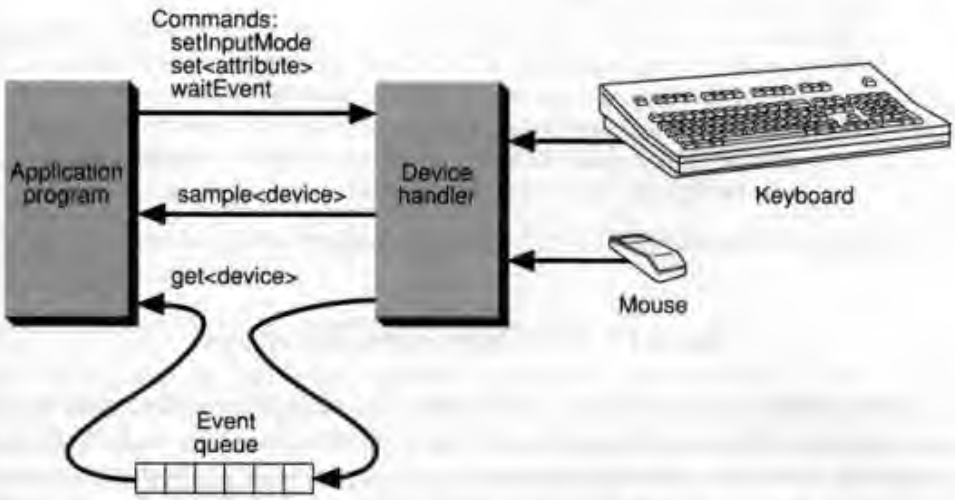


Fig. 2.16 Sampling versus event-handling using the event queue.

events in temporal order. In effect, the application specifies when it would like to be "interrupted."

When an application checks the event queue, it specifies whether it would like to enter a wait state. If the queue contains one or more event reports, the head event (representing the event that occurred earliest) is removed, and its information is made available to the application. If the queue is empty and a wait state is not desired, the application is informed that no event is available and it is free to continue execution. If the queue is empty and a wait state is desired, the application pauses until the next event occurs or until an application-specified maximum-wait-time interval passes. In effect, event mode replaces polling of the input devices with the much more efficient waiting on the event queue.

In summary, in sampling mode, the device is polled and an event measure is collected, regardless of any user activity. In event mode, the application either gets an event report from a prior user action or waits until a user action (or timeout) occurs. It is this "respond only when the user acts" behavior of event mode that is the essential difference between sampled and event-driven input. Event-driven programming may seem more complex than sampling, but you are already familiar with a similar technique used with the scanf function in a C program: C enables the keyboard, and the application waits in the scanf until the user has completed entering a line of text. You can access individual key-press events in C using the getc function.

Simple event-driven programs in SRGP or similar packages follow the reactive "ping-pong" interaction introduced in Section 1.6.4 and pseudocoded in Fig. 2.17; it can be nicely modeled as a finite-state automaton. More complex styles of interaction, allowing simultaneous program and user activity, are discussed in Chapters 8 through 10.

Event-driven applications typically spend most of their time in a wait state, since interaction is dominated by "think time" during which the user decides what to do next; even in fast-paced game applications, the number of events a user can generate in a second

Samsung/Google, EX1027, p. 63

```
initialize, including generating the initial image;
activate interactive device(s) in event mode;
while (user has not requested quit) {      /* main event loop */
    wait for user-triggered event on any of several devices;
    switch (device that caused event) {
        case DEVICE_1: collect DEVICE_1 event measure data, process, respond;
        case DEVICE_2: collect DEVICE_2 event measure data, process, respond;
        ...
    }
}
```

Fig. 2.17 Event-driven interaction scheme.

is a fraction of what the application could handle. Since SRGP typically implements event mode using true (hardware) interrupts, the wait state effectively uses no CPU time. On a multitasking system, the advantage is obvious: The event-mode application requires CPU time only for short bursts of activity immediately following user action, thereby freeing the CPU for other tasks.

One other point, about correct use of event mode, should be mentioned. Although the queueing mechanism does allow program and user to operate asynchronously, the user should not be allowed to get too far ahead of the program, because each event should result in an echo as well as some feedback from the application program. It is true that experienced users have learned to use "typeahead" to type in parameters such as file names or even operating-system commands while the system is processing earlier requests, especially if at least a character-by-character echo is provided immediately. In contrast, "mouseahead" for graphical commands is generally not as useful (and is much more dangerous), because the user usually needs to see the screen updated to reflect the application model's current state before the next graphical interaction.

2.2.4 Sample Mode

Activating, deactivating, and setting the mode of a device. The following procedure is used to activate or deactivate a device; it takes a device and a mode as parameters:

```
void SRGP_setInputMode (
        LOCATOR / KEYBOARD  inputDevice, INACTIVE / SAMPLE / EVENT  inputMode);
```

Thus, to set the locator to sample mode, we call

```
SRGP_setInputMode (LOCATOR, SAMPLE);
```

Initially, both devices are inactive. Placing a device in a mode in no way affects the other input device—both may be active simultaneously and even then need not be in the same mode.

The locator's measure. The locator is a logical abstraction of a mouse or data tablet, returning the cursor position as a screen (x, y) coordinate pair, the number of the button which most recently experienced a transition, and the state of the buttons as a *chord* array (since multiple buttons can be pressed simultaneously). The second field lets the application know which button caused the trigger for that event.

Samsung/Google, EX1027, p. 64

```
typedef struct {
    point position;
    int buttonOfMostRecentTransition;
    enum {UP, DOWN} buttonChord[MAX_BUTTON_COUNT];      /* Typically 1 to 3 */
} locatorMeasure;
```

Having activated the locator in sample mode with the SRGP_setInputMode procedure, we can ask its current measure using

```
void SRGP_sampleLocator (locatorMeasure *measure);
```

Let us examine the prototype sampling application shown in Fig. 2.18: a simple "painting" loop involving only button 1 on the locator. Such painting entails leaving a trail of paint where the user has dragged the locator while holding down this button; the locator is sampled in a loop as the user moves it. First, we must detect when the user starts painting by sampling the button until it is depressed; then, we place the paint (a filled rectangle in our simple example) at each sample point until the user releases the button.

The results of this sequence are crude: the paint rectangles are arbitrarily close together or far apart, with their density completely dependent on how far the locator was moved between consecutive samples. The sampling rate is determined essentially by the speed at which the CPU runs the operating system, the package, and the application.

Sample mode is available for both logical devices; however, the keyboard device is almost always operated in event mode, so techniques for sampling it are not addressed here.

2.2.5 Event Mode

Using event mode for initiation of sampling loop. Although the two sampling loops of the painting example (one to detect the button-down transition, the other to paint until

```
set up color/pattern attributes, and brush size in halfBrushHeight and halfBrushWidth;
SRGP_setInputMode (LOCATOR, SAMPLE);

/* First, sample until the button goes down. */
do {
    SRGP_sampleLocator (&locMeasure);
} while (locMeasure.buttonChord[0] == UP);

/* Perform the painting loop: */
/* Continuously place brush and then sample, until button is released. */
do {
    rect = SRGP_defRectangle (locMeasure.position.x - halfBrushWidth,
                              locMeasure.position.y - halfBrushHeight,
                              locMeasure.position.x + halfBrushWidth,
                              locMeasure.position.y + halfBrushHeight);
    SRGP_fillRectangle (rect);
    SRGP_sampleLocator (&locMeasure);
} while (locMeasure.buttonChord[0] == DOWN);
```

Fig. 2.18 Sampling loop for painting.

Samsung/Google, EX1027, p. 65

the button-up transition) certainly do the job, they put an unnecessary load on the CPU. Although this may not be a serious concern in a personal computer, it is not advisable in a system running multiple tasks, let alone doing time-sharing. Although it is certainly necessary to sample the locator repetitively for the painting loop itself (because we need to know the position of the locator at all times while the button is down), we do not need to use a sampling loop to wait for the button-down event that initiates the painting interaction. Event mode can be used when there is no need for measure information while waiting for an event.

SRGP_waitEvent. At any time after SRGP_setInputMode has activated a device in event mode, the program may inspect the event queue by entering the wait state with

```
inputDevice SRGP_waitEvent (int maxWaitTime);
```

The procedure returns immediately if the queue is not empty; otherwise, the first parameter specifies the maximum amount of time (measured in 1/60 second) for which the procedure should wait for an event if the queue is empty. A negative *maxWaitTime* (specified by the symbolic constant INDEFINITE) causes the procedure to wait indefinitely, whereas a value of zero causes it to return immediately, regardless of the state of the queue.

The identity of the device that issued the head event is returned in the *device* parameter. The special value NO_DEVICE is returned if no event was available within the specified time limit—that is, if the device timed out. The device type can then be tested to determine how the head event's measure should be retrieved (described later in this section).

The keyboard device. The trigger event for the keyboard device depends on the *processing mode* in which the keyboard device has been placed. EDIT mode is used when the application receives strings (e.g., file names, commands) from the user, who types and edits the string and then presses the Return key to trigger the event. In RAW mode, used for interactions in which the keyboard must be monitored closely, every key press triggers an event. The application uses the following procedure to set the processing mode.

```
void SRGP_setKeyboardProcessingMode (EDIT / RAW keyboardMode);
```

In EDIT mode, the user can type entire strings, correcting them with the backspace key as necessary, and then use the Return (or Enter) key as trigger. This mode is used when the user is to type in an entire string, such as a file name or a figure label. All control keys except backspace and Return are ignored, and the measure is the string as it appears at the time of the trigger. In RAW mode, on the other hand, each character typed, including control characters, is a trigger and is returned individually as the measure. This mode is used when individual keyboard characters act as commands—for example, for moving the cursor, for simple editing operations, or for video-game actions. RAW mode provides no echo, whereas EDIT mode echoes the string on the screen and displays a *text cursor* (such as an underscore or block character) where the next character to be typed will appear. Each backspace causes the text cursor to back up and to erase one character.

When SRGP_waitEvent returns the device code KEYBOARD, the application obtains the

Samsung/Google, EX1027, p. 66

measure associated with the event by calling

void SRGP_getKeyboard (**char** **measure*, **int** *measureSize*);

When the keyboard device is active in RAW mode, its measure is always exactly one character in length. In this case, the first character of the measure string returns the RAW measure.

The program shown in Fig. 2.19 demonstrates the use of EDIT mode. It receives a list of file names from the user, deleting each file so entered. When the user enters a null string (by pressing Return without typing any other characters), the interaction ends. During the interaction, the program waits indefinitely for the user to enter the next string.

Although this code explicitly specifies where the text prompt is to appear, it does not specify where the user's input string is typed (and corrected with the backspace). The location of this keyboard echo is specified by the programmer, as discussed in Section 2.2.7.

The locator device. The trigger event for the locator device is a press or release of a mouse button. When SRGP_waitEvent returns the device code LOCATOR, the application obtains the measure associated with the event by calling

void SRGP_getLocator (locatorMeasure **measure*);

Typically, the *position* field of the measure is used to determine in which area of the screen the user designated the point. For example, if the locator cursor is in a rectangular region where a menu button is displayed, the event should be interpreted as a request for some action; if it is in the main drawing area, the point might be inside a previously drawn object to indicate it should be selected, or in an "empty" region to indicate where a new object should be placed.

The pseudocode shown in Fig. 2.20 (similar to that shown previously for the keyboard) implements another use of the locator, letting the user specify points at which markers are to be placed. The user exits the marker-placing loop by pressing the locator button while the cursor points to a screen button, a rectangle containing the text "quit."

In this example, only the user's pressing of locator button 1 is significant; releases of the button are ignored. Note that the button must be released before the next button-press event can take place—the event is triggered by a transition, not by a button state. Furthermore, to ensure that events coming from the other buttons do not disturb this

```
SRGP_setInputMode (KEYBOARD, EVENT);      /* Assume only the keyboard is active. */
SRGP_setKeyboardProcessingMode (EDIT);
pt = SRGP_defPoint (100, 100);
SRGP_text (pt, "Specify one or more files to be deleted; to exit, press Return.");

/* main event loop */
do {
    device = SRGP_waitEvent (INDEFINITE);
    SRGP_getKeyboard (measure, measureSize);
    if (*measure != NULL)
        DeleteFile (measure);                 /* DeleteFile does confirmation, etc. */
} while (*measure != NULL);
```

Fig. 2.19 EDIT-mode keyboard interaction.

Samsung/Google, EX1027, p. 67

```
const int QUIT_BUTTON = 0, QUIT_MASK = 0x1;

create the on-screen Quit button;
SRGP_setLocatorButtonMask (QUIT_MASK);
SRGP_setInputMode (LOCATOR, EVENT);      /* Assume only locator is active. */

/* main event loop */
terminate = FALSE;
while (!terminate) {
    device = SRGP_waitEvent (INDEFINITE);
    SRGP_getLocator (&measure);
    if (measure.buttonChord[QUIT_BUTTON] == DOWN) {
        if (PickedQuitButton (measure.position))
            terminate = TRUE;
        else
            SRGP_marker (measure.position);
    }
}
```

Fig. 2.20 Locator interaction.

interaction, the application tells SRGP which buttons are to trigger a locator event by calling

```
void SRGP_setLocatorButtonMask (int activeButtons);
```

The default locator-button mask is set to one, but no matter what the mask is, all buttons always have a measure. On implementations that support fewer than three buttons, references to any nonexistent buttons are simply ignored by SRGP, and these buttons' measures always contain UP.

The function PickedQuitButton compares the measure position against the bounds of the quit button rectangle and returns a Boolean value signifying whether or not the user picked the quit button. This process is a simple example of *pick correlation*, as discussed in in the next section.

Waiting for multiple events. The code fragments in Figs. 2.19 and 2.20 did not illustrate event mode's greatest advantage: the ability to wait for more than one device at the same time. SRGP queues events of enabled devices in chronological order and lets the application program take the first one off the queue when SRGP_waitEvent is called. Unlike hardware interrupts, which are processed in order of priorities, events are thus processed strictly in temporal order. The application examines the returned device code to determine which device caused the event.

The procedure shown in Fig. 2.21 allows the user to place any number of small circle markers anywhere within a rectangular drawing area. The user places a marker by pointing to the desired position and pressing button 1; she requests that the interaction be terminated by either pressing button 3 or typing "q" or "Q."

2.2.6 Pick Correlation for Interaction Handling

A graphics application customarily divides the screen area into regions dedicated to specific purposes. When the user presses the locator button, the application must determine exactly

Samsung/Google, EX1027, p. 68

```
const int PLACE_BUTTON = 0, PLACE_MASK = 0x1,
          QUIT_BUTTON = 2, QUIT_MASK = 0x4;

generate initial screen layout;
SRGP_setInputMode (KEYBOARD, EVENT);
SRGP_setKeyboardProcessingMode (RAW);
SRGP_setInputMode (LOCATOR, EVENT);
SRGP_setLocatorButtonMask (PLACE_MASK | QUIT_MASK);      /* Ignore middle button. */

/* Main event loop */
terminate = FALSE;
while (!terminate) {
    device = SRGP_waitEvent (INDEFINITE);
    switch (device) {
        case KEYBOARD:
            SRGP_getKeyboard (keyMeasure, keyMeasureSize);
            terminate = (keyMeasure[0] == 'q') || (keyMeasure[0] == 'Q');
            break;
        case LOCATOR:
            SRGP_getLocator (&locMeasure);
            switch (locMeasure.buttonOfMostRecentTransition) {
                case PLACE_BUTTON:
                    if ((locMeasure.buttonChord[PLACE_BUTTON] == DOWN)
                        && InDrawingArea (locMeasure.position))
                        SRGP_marker (locMeasure.position);
                    break;
                case QUIT_BUTTON:
                    terminate = TRUE;
                    break;
            } /* button switch */
    } /* device switch */
} /* while */
```

Fig. 2.21 Use of several devices simultaneously.

what screen button, icon, or other object was selected, if any, so that it can respond appropriately. This determination, called *pick correlation*, is a fundamental part of interactive graphics.

An application program using SRGP performs pick correlation by determining in which region the cursor is located, and then which object within that region, if any, the user is selecting. Points in an empty subregion might be ignored (if the point is between menu buttons in a menu, for example) or might specify the desired position for a new object (if the point lies in the main drawing area). Since a great many regions on the screen are upright rectangles, almost all the work for pick correlation can be done by a simple, frequently used Boolean function that checks whether a given point lies in a given rectangle. The GEOM package distributed with SRGP includes this function (GEOM_ptInRect) as well as other utilities for coordinate arithmetic. (For more information on pick correlation, see Section 7.12.)

Samsung/Google, EX1027, p. 69

Let us look at a classic example of pick correlation. Consider a painting application with a *menu bar* across the top of the screen. This menu bar contains the names of pull-down menus, called menu *headers*. When the user picks a header (by placing the cursor on top of the header's text string and pressing a locator button), the corresponding *menu body* is displayed on the screen below the header and the header is highlighted. After the user selects an entry on the menu (by releasing the locator button),[6] the menu body disappears and the header is unhighlighted. The rest of the screen contains the main drawing area in which the user can place and pick objects. The application, in creating each object, has assigned it a unique identifier (ID) that is returned by the pick-correlation procedure for further processing of the object.

When a point is obtained from the locator via a button-down event, the high-level interaction-handling schema shown in Fig. 2.22 is executed; it is essentially a dispatching procedure that uses pick correlation within the menu bar or the main drawing area to divide the work among menu- and object-picking procedures. First, if the cursor was in the menu bar, a subsidiary correlation procedure determines whether the user selected a menu header. If so, a procedure (detailed in Section 2.3.1) is called to perform the menu interaction; it returns an index specifying which item within the menu's body (if any) was chosen. The menu ID and item index together uniquely identify the action that should be taken in response. If the cursor was not in the menu bar but rather in the main drawing area, another subsidiary correlation procedure is called to determine what object was picked, if any. If an object was picked, a processing procedure is called to respond appropriately.

The procedure CorrelateMenuBar performs a finer correlation by calling GEOM_pointInRect once for each menu header in the menu bar; it accesses a database storing the rectangular screen extent of each header. The procedure CorrelateDrawingArea must do more sophisticated correlation because, typically, objects in the drawing area may overlap and are not necessarily rectangular.

2.2.7 Setting Device Measure and Attributes

Each input device has its own set of attributes, and the application can set these attributes to custom-tailor the feedback the device presents to the user. (The button mask presented earlier is also an attribute; it differs from those presented here in that it does not affect feedback.) Like output-primitive attributes, input-device attributes are set modally by specific procedures. Attributes can be set at any time, whether or not the device is active.

In addition, each input device's measure, normally determined by the user's actions, can also be set by the application. Unlike input-device attributes, an input device's measure is reset to a default value when the device is deactivated; thus, upon reactivation, devices initially have predictable values, a convenience to the programmer and to the user. This automatic resetting can be overridden by explicitly setting a device's measure while it is inactive.

Locator echo attributes. Several types of echo are useful for the locator. The

[6] This sequence, corresponding to the Macintosh menu-interaction style, is only one of many different ways the user interface could be designed.

Samsung/Google, EX1027, p. 70

```
void HighLevelInteractionHandler (locatorMeasure measureOfLocator)
{
    if (GEOM_pointInRect (measureOfLocator.position, menuBarExtent)) {
        /* Find out which menu's header, if any, the user selected. */
        /* Then, pull down that menu's body. */
        menuID = CorrelateMenuBar (measureOfLocator.position);
        if (menuID > 0) {
            chosenItemIndex = PerformPulldownMenuInteraction (menuID);
            if (chosenItemIndex > 0)
                PerformActionChosenFromMenu (menuID, chosenItemIndex);
        }
    } else {        /* The user picked within the drawing area; find out what and respond. */
        objectID = CorrelateDrawingArea (measureOfLocator.position);
        if (objectID > 0)
            ProcessObject (objectID);
    }
}  /* HighLevelInteractionHandler */
```

Fig. 2.22 High-level interaction scheme for menu handling.

programmer can control both echo type and cursor shape with

```
void SRGP_setLocatorEchoType (
    NO_ECHO / CURSOR / RUBBER_LINE / RUBBER_RECT echoType);
```

The default is CURSOR, and SRGP implementations supply a cursor table from which an application selects a desired cursor shape (see the reference manual). A common use of the ability to specify the cursor shape dynamically is to provide feedback by changing the cursor shape according to the region in which the cursor lies. RUBBER_LINE and RUBBER_RECT echo are commonly used to specify a line or box. With these set, SRGP automatically draws a continuously updated line or rectangle as the user moves the locator. The line or rectangle is defined by two points, the anchor point (another locator attribute) and the current locator position. Figure 2.23 illustrates the use of these two modes for user specification of a line and a rectangle.

In Fig. 2.23(a), the echo is a cross-hair cursor, and the user is about to press the locator button. The application initiates a rubber echo, anchored at the current locator position, in response to the button press. In parts (b) and (c), the user's movement of the locator device is echoed by the rubber primitive. The locator position in part (c) is returned to the application when the user releases the button, and the application responds by drawing a line or rectangle primitive and restoring normal cursor echo (see part d).

The anchor point for rubber echo is set with

```
void SRGP_setLocatorEchoRubberAnchor (point position);
```

An application typically uses the *position* field of the measure obtained from the most recent locator-button–press event as the anchor position, since that button press typically initiates the rubber-echo sequence.

Locator measure control. The *position* portion of the locator measure is automatically

Samsung/Google, EX1027, p. 71

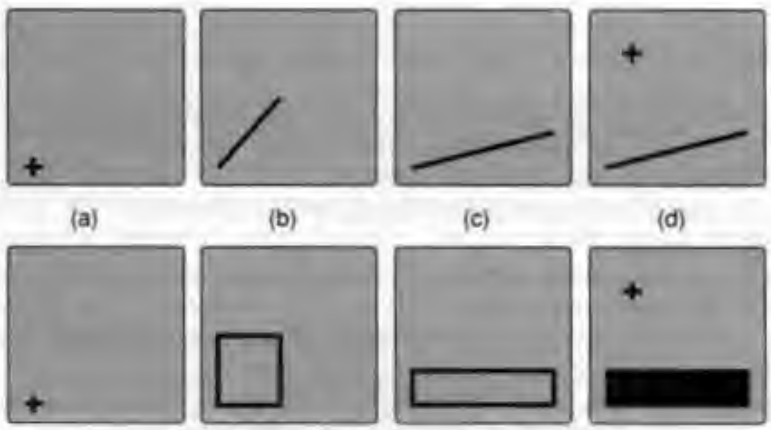


Fig. 2.23 Rubber-echo scenarios.

reset to the center of the screen whenever the locator is deactivated. Unless the programmer explicitly resets it, the measure (and feedback position, if the echo is active) is initialized to that same position when the device is reactivated. At any time, whether the device is active or inactive, the programmer can reset the locator's measure (the *position* portion, not the fields concerning the buttons) using

void SRGP_setLocatorMeasure (point *position*);

Resetting the measure while the locator is inactive has no immediate effect on the screen, but resetting it while the locator is active changes the echo (if any) accordingly. Thus, if the program wants the cursor to appear initially at a position other than the center when the locator is activated, a call to SRGP_setLocatorMeasure with that initial position must precede the call to SRGP_setInputMode. This technique is commonly used to achieve continuity of cursor position: The last measure before the locator was deactivated is stored, and the cursor is returned to that position when it is reactivated.

Keyboard attributes and measure control. Unlike the locator, whose echo is positioned to reflect movements of a physical device, there is no obvious screen position for a keyboard device's echo. The position is thus an attribute (with an implementation-specific default value) of the keyboard device that can be set via

void SRGP_setKeyboardEchoOrigin (point *origin*);

The default measure for the keyboard is automatically reset to the null string when the keyboard is deactivated. Setting the measure explicitly to a nonnull initial value just before activating the keyboard is a convenient way to present a default input string (displayed by SRGP as soon as echoing begins) that the user can accept as is or modify before pressing the Return key, thereby minimizing typing. The keyboard's measure is set via

void SRGP_setKeyboardMeasure (**char** **measure*);

2.3 RASTER GRAPHICS FEATURES

By now, we have introduced most of the features of SRGP. This section discusses the remaining facilities that take particular advantage of raster hardware, especially the ability

Samsung/Google, EX1027, p. 72

to save and restore pieces of the screen as they are overlaid by other images, such as windows or temporary menus. Such image manipulations are done under control of window- and menu-manager application programs. We also introduce offscreen bitmaps (called *canvases*) for storing windows and menus, and we discuss the use of clipping rectangles.

2.3.1 Canvases

The best way to make complex icons or menus appear and disappear quickly is to create them once in memory and then to copy them onto the screen as needed. Raster graphics packages do this by generating the primitives in invisible, offscreen bitmaps or pixmaps of the requisite size, called *canvases* in SRGP, and then copying the canvases to and from display memory. This technique is, in effect, a type of buffering. Moving blocks of pixels back and forth is faster, in general, than is regenerating the information, given the existence of the fast SRGP_copyPixel operation that we shall discuss soon.

An SRGP *canvas* is a data structure that stores an image as a 2D array of pixels. It also stores some control information concerning the size and attributes of the image. Each canvas represents its image in its own Cartesian coordinate system, which is identical to that of the screen shown in Fig. 2.1; in fact, the screen is itself a canvas, special solely in that it is the only canvas that is displayed. To make an image stored in an off-screen canvas visible, the application must copy it onto the screen canvas. Beforehand, the portion of the screen image where the new image—for example, a menu—will appear can be saved by copying the pixels in that region to an offscreen canvas. When the menu selection has taken place, the screen image is restored by copying back these pixels.

At any given time, there is one *currently active* canvas: the canvas into which new primitives are drawn and to which new attribute settings apply. This canvas may be the screen canvas (the default we have been using) or an offscreen canvas. The coordinates passed to the primitive procedures are expressed in terms of the local coordinate space of the currently active canvas. Each canvas also has its own complete set of SRGP attributes, which affect all drawing on that canvas and are set to the standard default values when the canvas is created. Calls to attribute-setting procedures modify only the attributes in the currently active canvas. It is convenient to think of a canvas as a virtual screen of program-specified dimensions, having its own associated pixmap, coordinate system, and attribute group. These properties of the canvas are sometimes called the *state* or *context* of the canvas.

When SRGP is initialized, the *screen canvas* is automatically created and made active. All our programs thus far have generated primitives into only that canvas. It is the only canvas visible on the screen, and its ID is SCREEN_CANVAS, an SRGP constant. A new offscreen canvas is created by calling the following procedure, which returns the ID allocated for the new canvas:

```
canvasID SRGP_createCanvas (int width, int height);
```

Like the screen, the new canvas's local coordinate system origin (0, 0) is at the bottom-left corner and the top-right corner is at (*width*–1, *height*–1). A 1 by 1 canvas is therefore defined by width and height of 1, and its bottom-left and top-right corners are both (0, 0)! This is consistent with our treatment of pixels as being at grid intersections: The single pixel in a 1 by 1 canvas is at (0, 0).

Samsung/Google, EX1027, p. 73

A newly created canvas is automatically made active and its pixels are initialized to color 0 (as is also done for the screen canvas before any primitives are displayed). Once a canvas is created, its size cannot be changed. Also, the programmer cannot control the number of bits per pixel in a canvas, since SRGP uses as many bits per pixel as the hardware allows. The attributes of a canvas are kept as part of its "local" state information; thus, the program does not need to save the currently active canvas's attributes explicitly before creating a new active canvas.

The application selects a previously created canvas to be the currently active canvas via

```
void SRGP_useCanvas (canvasID id);
```

A canvas being activated in no way implies that that canvas is made visible; an image in an offscreen canvas must be copied onto the screen canvas (using the SGRP_copyPixel procedure described shortly) in order to be seen.

Canvases are deleted by the following procedure, which may not be used to delete the screen canvas or the currently active canvas.

```
void SRGP_deleteCanvas (canvasID id);
```

The following procedures allow inquiry of the size of a canvas; one returns the rectangle which defines the canvas coordinate system (the bottom-left point always being (0, 0)), and the other returns the width and height as separate quantities.

```
rectangle SRGP_inquireCanvasExtent (canvasID id);
void SRGP_inquireCanvasSize (canvasID id, int *width, int *height);
```

Let us examine the way canvases can be used for the implementation of PerformPulldownMenuInteraction, the procedure called by the high-level interaction handler presented in Fig. 2.22 and Section 2.2.6. The procedure is implemented by the pseudocode of Fig. 2.24, and its sequence of actions is illustrated in Fig. 2.25. Each menu has a unique

```
int PerformPulldownMenuInteraction (int menuID);
/* The saving/copying of rectangular regions of canvases is described in Section 2.3.3. */
{
    highlight the menu header in the menu bar;
    menuBodyScreenExtent = screen-area rectangle at which menu body should appear;
    save the current pixels of the menuBodyScreenExtent in a temporary canvas;
        /* See Fig. 2.25a. */
    copy menu body image from body canvas to menuBodyScreenExtent;
        /* See Fig. 2.25b and C code in Fig. 2.28. */
    wait for button-up signaling the user made a selection, then get locator measure;
    copy saved image from temporary canvas back to menuBodyScreenExtent;
        /* See Fig. 2.25c */
    if (GEOM_pointInRect (measureOfLocator.position, menuBodyScreenExtent))
        calculate and return index of chosen item, using y coord of measure position;
    else
        return 0;
} /* PerformPulldownMenuInteraction */
```

Fig. 2.24 Pseudocode for PerformPulldownMenuInteraction.

Samsung/Google, EX1027, p. 74

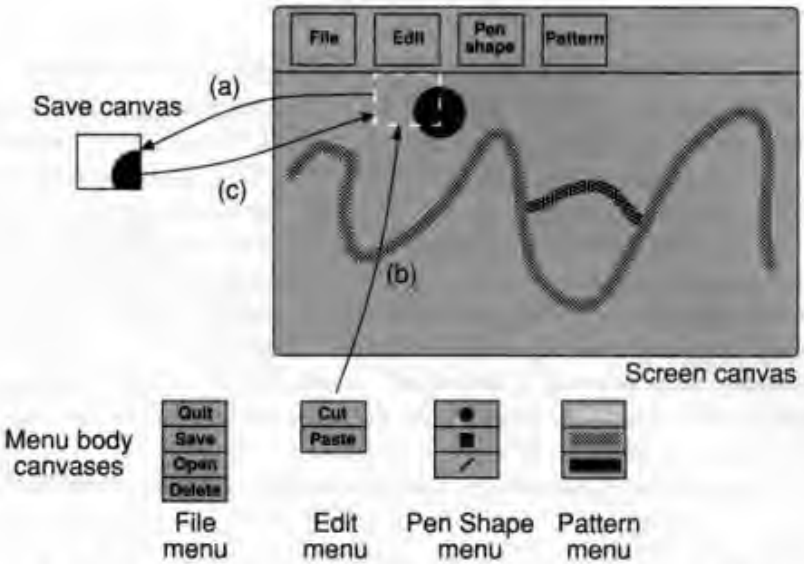


Fig. 2.25 Saving and restoring area covered by menu body.

ID (returned by the CorrelateMenuBar function) that can be used to locate a database record containing the following information about the appearance of the menu body:

- The ID of the canvas storing the menu's body
- The rectangular area (called *menuBodyScreenExtent* in the pseudocode), specified in screen-canvas coordinates, in which the menu's body should appear when the user pulls down the menu by clicking in its header

2.3.2 Clipping Rectangles

Often, it is desirable to restrict the effect of graphics primitives to a subregion of the active canvas, to protect other portions of the canvas. To facilitate this, SRGP maintains a *clip rectangle* attribute. All primitives are clipped to the boundaries of this rectangle; that is, primitives (or portions of primitives) lying outside the clip rectangle are not drawn. Like any attribute, the clip rectangle can be changed at any time, and its most recent setting is stored with the canvas's attribute group. The default clipping rectangle (what we have used so far) is the full canvas; it can be changed to be smaller than the canvas, but it cannot extend beyond the canvas boundaries. The relevant set and inquiry calls for the clip rectangle are

```
void SRGP_setClipRectangle (rectangle clipRect);
rectangle SRGP_inquireClipRectangle (void);
```

A painting application like that presented in Section 2.2.4 would use the clip rectangle to restrict the placement of paint to the drawing region of the screen, ensuring that the surrounding menu areas are not damaged. Although SRGP offers only a single upright rectangle clipping boundary, some more sophisticated software such as POSTSCRIPT offers multiple, arbitrarily shaped clipping regions.

Samsung/Google, EX1027, p. 75

2.3.3 The SRGP_copyPixel Operation

The powerful SRGP_copyPixel command is a typical raster command that is often called bitBlt (bit block transfer) or pixBlt (pixel Blt) when implemented directly in hardware; it first became available in microcode on the pioneering ALTO bitmap workstation at Xerox Palo Alto Research Center in the early 1970s [INGA81]. This command is used to copy an array of pixels from a rectangular region of a canvas, the *source* region, to a *destination* region in the currently active canvas (see Fig. 2.26). The SRGP facility provides only restricted functionality in that the destination rectangle must be of the same size as the source. In more powerful versions, the source can be copied to a destination region of a different size, being automatically scaled to fit (see Chapter 19). Also, additional features may be available, such as *masks* to selectively shield desired source or destination pixels from copying (see Chapter 19), and *halftone patterns* that can be used to "screen" (i.e., shade) the destination region.

SRGP_copyPixel can copy between any two canvases and is specified as follows:

```
void SRGP_copyPixel (
    canvasID sourceCanvas, rectangle sourceRect, point destCorner);
```

The *sourceRect* specifies the source region in an arbitrary canvas, and *destCorner* specifies the bottom-left corner of the destination rectangle inside the currently active canvas, each in their own coordinate systems. The copy operation is subject to the same clip rectangle that prevents primitives from generating pixels into protected regions of a canvas. Thus, the region into which pixels are ultimately copied is the intersection of the extent of the destination canvas, the destination region, and the clip rectangle, shown as the striped region in Fig. 2.27.

To show the use of copyPixel in handling pull-down menus, let us implement the fourth statement of pseudocode—"copy menu body image"—from the PerformPulldownMenu-Interaction function (Fig. 2.24). In the third statement of the pseudocode, we saved in an offscreen canvas the screen region where the menu body is to go; now, we wish to copy the menu body to the screen.

The Pascal code is shown in Fig. 2.28. We must be sure to distinguish between the two rectangles that are of identical size but that are expressed in different coordinate systems.

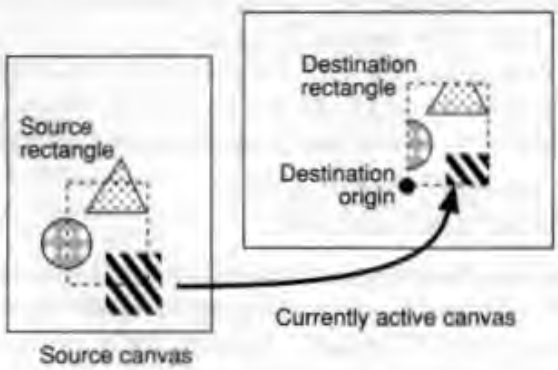


Fig. 2.26 SRGP_copyPixel.

Samsung/Google, EX1027, p. 76

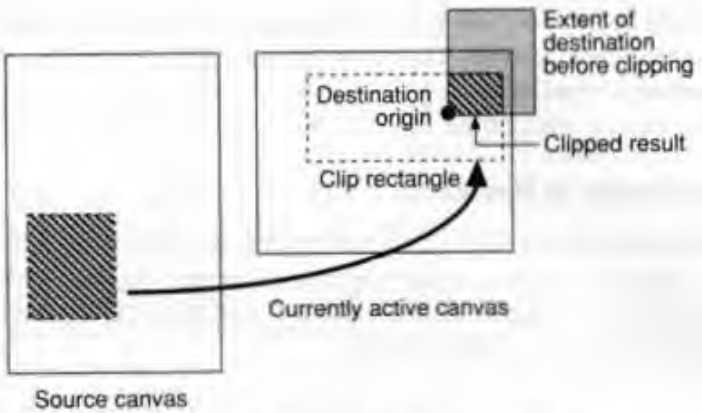


Fig. 2.27 Clipping during copyPixel.

The first rectangle, which we call *menuBodyExtent* in the code, is simply the extent of the menu body's canvas in its own coordinate system. This extent is used as the source rectangle in the SRGP_copyPixel operation that puts the menu on the screen. The *menuBodyScreenExtent* is a rectangle of the same size that specifies in screen coordinates the position in which the menu body should appear; that extent's bottom-left corner is horizontally aligned with the left side of the menu header, and its top-right corner abuts the bottom of the menu bar. (Figure 2.25 symbolizes the Edit menu's screen extent as a dotted outline, and its body extent as a solid outline.) The *menuBodyScreenExtent*'s bottom-left point is used to specify the destination for the SRGP_copyPixel that copies the menu body. It is also the source rectangle for the initial save of the screen area to be overlaid by the menu body and the destination of the final restore.

```
/* This code fragment copies a menu-body image onto screen,  */
/* at the screen position stored in the body's record.  */

/* Save the ID of the currently active canvas.  */
saveCanvasID = SRGP_inquireActiveCanvas();

/* Save the screen canvas' clip-rectangle attribute value.  */
SRGP_useCanvas (SCREEN_CANVAS);
saveClipRectangle = SRGP_inquireClipRectangle ();

/* Temporarily set screen clip rectangle to allow writing to all of the screen.  */
SRGP_setClipRectangle (SCREEN_EXTENT);

/* Copy menu body from its canvas to its proper area below the header in the menu bar.  */
SRGP_copyPixel (menuCanvasID, menuBodyExtent, menuBodyScreenExtent.bottomLeft);

/* Restore screen attributes and active canvas.  */
SRGP_setClipRectangle (saveClipRectangle);
SRGP_useCanvas (saveCanvasID);
```

Fig. 2.28 Code for copying the menu body to the screen.

Samsung/Google, EX1027, p. 77

Notice that the application's state is saved and restored to eliminate side effects. We set the screen clip rectangle to SCREEN_EXTENT before copying; alternatively, we could set it to the exact *menuBodyScreenExtent*.

2.3.4 Write Mode or RasterOp

SRGP_copyPixel can do more than just move an array of pixels from a source region to a destination. It can also execute a logical (bitwise) operation between each corresponding pair of pixels in the source and destination regions, then place the result in the destination region. This operation can be symbolized as

$$D \leftarrow S \textbf{ op } D$$

where **op**, frequently called the *RasterOp* or *write mode*, consists in general of the 16 Boolean operators. Only the most common of these—**replace**, **or**, **xor**, and **and**—are supported by SRGP; these are shown for a 1-bit-per-pixel image in Fig. 2.29.

Write mode affects not only SRGP_copyPixel, but also any new primitives written onto a canvas. As each pixel (either of a source rectangle of a SRGP_copyPixel or of a primitive) is stored in its memory location, either it is written in destructive **replace** mode or its value is logically combined with the previously stored value of the pixel. (This bitwise combination of source and destination values is similar to the way a CPU's hardware performs arithmetic or logical operations on the contents of a memory location during a read–modify–write memory cycle.) Although **replace** is by far the most common mode, **xor** is quite useful for generating dynamic objects, such as cursors and rubberband echoes, as we discuss shortly.

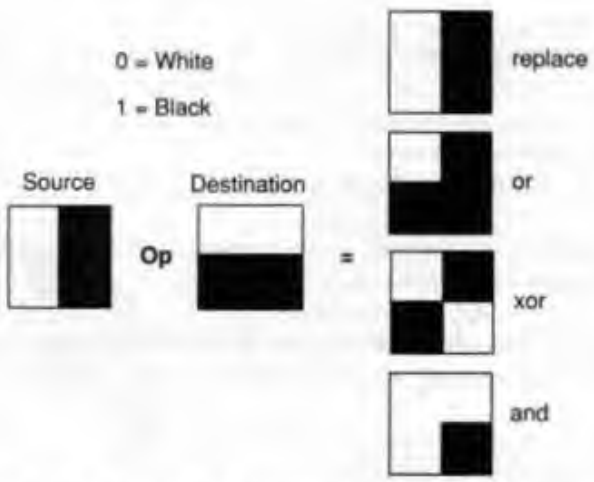


Fig. 2.29 Write modes for combining source and destination pixels.

Samsung/Google, EX1027, p. 78

We set the write-mode attribute with:

```
void SRGP_setWriteMode (
    WRITE_REPLACE / WRITE_XOR / WRITE_OR / WRITE_AND writeMode);
```

Since all primitives are generated according to the current write mode, the SRGP programmer must be sure to set this mode explicitly and not to rely on the default setting of WRITE_REPLACE.

To see how RasterOp works, we look at how the package actually stores and manipulates pixels; this is the only place where hardware and implementation considerations intrude on the abstract view of raster graphics that we have maintained so far.

RasterOps are performed on the pixel values, which are indices into the color table, not on the hardware color specifications stored as entries in the color table. Thus, for a bilevel, 1-bit-per-pixel system, the RasterOp is done on two indices of 1 bit each. For an 8-bit-per-pixel color system, the RasterOp is done as a bitwise logical operation on two 8-bit indices.

Although the interpretation of the four basic operations on 1-bit-per-pixel monochrome images shown in Fig. 2.29 is natural enough, the results of all but **replace** mode are not nearly so natural for n-bit-per-pixel images ($n > 1$), since a bitwise logical operation on the source and destination indices yields a third index whose color value may be wholly unrelated to the source and destination colors.

The **replace** mode involves writing over what is already on the screen (or canvas). This destructive write operation is the normal mode for drawing primitives, and is customarily used to move and pop windows. It can also be used to "erase" old primitives by drawing over them in the application screen background pattern.

The **or** mode on bilevel displays makes a nondestructive addition to what is already on the canvas. With color 0 as white background and color 1 as black foreground, **or**ing a gray fill pattern onto a white background changes the underlying bits to show the gray pattern. But **or**ing the gray pattern over a black area has no effect on the screen. Thus, **or**ing a light-gray paint swath over a polygon filled with a brick pattern merely fills in the bricks with the brush pattern; it does not erase the black edges of the bricks, as **replace** mode would. Painting is often done in **or** mode for this reason (see Exercise 2.7).

The **xor** mode on bilevel displays can be used to invert a destination region. For example, to highlight a button selected by the user, we set **xor** mode and generate a filled rectangle primitive with color 1, thereby toggling all pixels of the botton: 0 **xor** 1 = 1, 1 **xor** 1 = 0. To restore the button's original status, we simply stay in **xor** mode and draw the rectangle a second time, thereby toggling the bits back to their original state. This technique is also used internally by SRGP to provide the locator's rubber-line and rubber-rectangle echo modes (see Exercise 2.4).

On many bilevel graphics displays, the **xor** technique is used by the underlying hardware (or in some cases software) to display the locator's cursor image in a nondestructive manner. There are some disadvantages to this simple technique; when the cursor is on top of a background with a fine pattern that is almost 50 percent black and 50 percent white, it is possible for the cursor to be only barely noticeable. Therefore, many

Samsung/Google, EX1027, p. 79

bilevel displays and most color displays use **replace** mode for the cursor echo; this technique complicates the echo hardware or software (see Exercise 2.5).

The **and** mode can be used, for example, to reset pixels selectively in the destination region to color 0.

2.4 LIMITATIONS OF SRGP

Although SRGP is a powerful package supporting a large class of applications, inherent limitations make it less than optimal for some applications. Most obviously, SRGP provides no support for applications displaying 3D geometry. There are also more subtle limitations that affect even many 2D applications:

- The machine-dependent integer coordinate system of SRGP is too inflexible for those applications that require the greater precision, range, and convenience of floating-point.
- SRGP stores an image in a canvas in a semantics-free manner as a matrix of unconnected pixel values rather than as a collection of graphics objects (primitives), and thus does not support object-level operations, such as "delete," "move," "change color." Because SRGP keeps no record of the actions that produced the current screen image, it also cannot refresh a screen if the image is damaged by other software, nor can it re–scan-convert the primitives to produce an image for display on a device with a different resolution.

2.4.1 Application Coordinate Systems

In the previous chapter, we introduced the notion that, for most applications, drawings are only a means to an end, and that the primary role of the application database is to support such processes as analysis, simulation, verification, and manufacturing. The database must therefore store geometric information using the range and precision required by these processes, independent of the coordinate system and resolution of the display device. For example, a VLSI CAD/CAM program may need to represent circuits that are 1 to 2 centimeters (cm) long at a precision of half a micron, whereas an astronomy program may need a range of 1 to 10^9 light-years with a precision of a million miles. For maximum flexibility and range, many applications use floating-point *world coordinates* for storing geometry in their database.

Such an application could do the mapping from world to device coordinates itself; however, considering the complexity of this mapping (which we shall discuss in Chapter 6), it is convenient to use a graphics package that accepts primitives specified in world coordinates and maps them to the display device in a machine-independent manner. The recent availability of inexpensive floating-point chips offering roughly the performance of integer arithmetic has significantly reduced the time penalty associated with the use of floating-point—the flexibility makes it well worth its cost to the applications that need it.

For 2D graphics, the most common software that provides floating-point coordinates is Adobe's PostScript (see Chapter 19), used both as the standard page-description language for driving hardcopy printers and (in an extension called Display PostScript) as the graphics

Samsung/Google, EX1027, p. 80

package for windowing systems on some workstations. For 3D floating-point graphics, PHIGS and PHIGS+ are now widely available, and various 3D extensions to PostScript are appearing.

2.4.2 Storage of Primitives for Respecification

Consider what happens when an application using SRGP needs to redraw a picture at a different size, or at the same size on a display device with a different resolution (such as a higher-resolution printer). Because SRGP has no record of the primitives it has drawn, the application must respecify the entire set of primitives to SRGP after scaling the coordinates.

If SRGP were enhanced to retain a record of all specified primitives, the application could let SRGP regenerate them from its own storage. SRGP could then support another commonly needed operation, refreshing the screen. On some graphics systems, the application's screen image can be damaged by messages from other users or applications; unless the screen canvas can be refreshed from a redundantly stored copy in an offscreen canvas, respecification of the primitives is the only way to repair the damage.

The most important advantage of having the package store primitives is the support of editing operations that are the essence of drawing or construction applications, a class of programs that is quite different from the painting applications illustrated in this chapter's examples. A *painting program* allows the user to paint arbitrary swaths using a brush of varying size, shape, color, and pattern. More complete painting programs also allow placement of such predefined shapes as rectangles, polygons, and circles. Any part of the canvas can be subsequently edited at a pixel level; portions of an object can be covered with paint, or arbitrary rectangular regions of the canvas can be copied or moved elsewhere. The user cannot point to a previously drawn shape or to a painted swath and then delete or move it as a coherent, indivisible object. This limitation exists because a painting program allows an object, once placed on the canvas, to be mutilated and fragmented, losing its identity as a coherent object. For example, what would it mean for the user to point to a fragment of an object that had been split into pieces that were independently positioned in various areas of the screen? Would the user be referring to the fragment itself, or to the entire original object? In essence, the ability to affect individual pixels makes pick correlation—and therefore object picking and editing—impossible.

A *drawing program*, conversely, allows the user to pick and edit any object at any time. These applications, also called *layout editors* or *graphical illustrators*, allow a user to position standard shapes (also called *symbols*, *templates*, or *objects*) and then to edit the layout by deleting, moving, rotating, and scaling these shapes. Similar interactive programs that allow users to assemble complex 3D objects from simpler ones are called *geometric editors* or *construction programs*.

Scaling, screen refreshing, and object-level editing all require the storage and respecification of primitives by the application or by the graphics package. If the application stores the primitives, it can perform the respecification; however, these operations are more complex than they may seem at first glance. For example, a primitive can be deleted trivially by erasing the screen and respecifying all the primitives (except, of course, the deleted one); however, a more efficient method is to erase the primitive's image by drawing

Samsung/Google, EX1027, p. 81

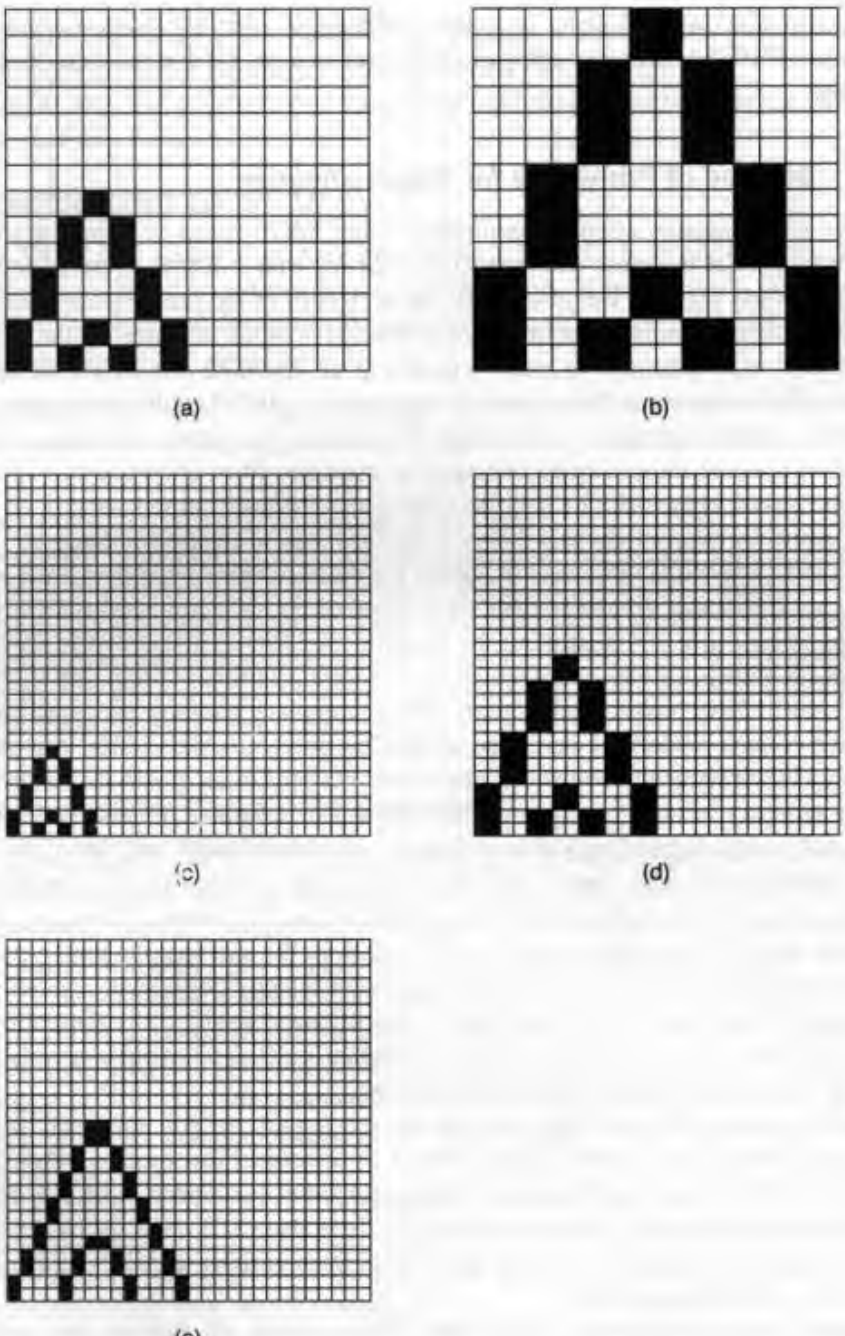


Fig. 2.30 The effects of pixel replication. (a) Original image at screen resolution. (b) Zoomed (2×) image at screen resolution. (c) Original image printed on device with twice the screen's resolution. (d) Zoomed image on same device as (c), using pixel replication to maintain image size. (e) Original image printed on same device as (c), using re–scan-conversion to maintain image size.

Samsung/Google, EX1027, p. 82

the application screen background on top of it and then to respecify any primitives that may have been damaged. Because these operations are both complex and frequently needed, there is good reason for moving their functionality into the graphics package itself.

An object-level, geometric graphics package, such as GKS or PHIGS, lets the application define objects using a 2D or 3D floating-point coordinate system. The package stores objects internally, allows the application to edit the stored objects, and updates the screen whenever necessary due to an editing operation. The package also performs pick correlation, producing an object ID when given a screen coordinate. Because these packages manipulate objects, they cannot permit pixel-level manipulations (copyPixel and write mode)—this is the price of preserving object coherence. Thus, neither a raster graphics package without primitive storage nor a geometric graphics package with primitive storage satisfies all needs. Chapter 7 discusses the pros and cons of the retention of primitives in the graphics package.

Image scaling via pixel replication. If neither the application nor the package has a record of the primitives (as is typical of most painting programs), scaling cannot be done by respecifying the primitives with scaled endpoint coordinates. All that can be done is to scale the contents of the canvas using read-pixel and write-pixel operations. The simple, fast way to scale up a bitmap/pixmap image (to make it larger) is via *pixel replication*, as shown in Fig. 2.30(a,b); here, each pixel is replaced by an N by N block of pixels, thus enlarging the image by a scale factor of N.

With pixel replication, the image becomes larger, but it also becomes coarser, since no new information is provided beyond that contained in the original pixel-level representation (compare Fig. 2.30a to Fig. 2.30b). Moreover, pixel replication can increase an image's size by only an integer factor. We must use a second technique—area sampling and filtering (discussed in Chapters 3, 14, 17, and 19)—to scale up or down properly. Filtering works best on pixmaps with depth > 1.

The problem of image scaling arises frequently, particularly when an image created by a painting program is to be printed. Let us consider sending a canvas to a printer that provides twice the resolution of the screen. Each pixel is now one-half its original size; thus, we can show the original image with the same number of pixels at half the size (Fig. 2.30c), or we can use pixel replication to produce an image of the original size without taking advantage of the finer resolution of the printer (Fig. 2.30d). Either way, something is lost, size or quality, and the only scaling method that does not sacrifice quality is respecification (Fig. 2.30e).

2.5 SUMMARY

In this chapter, we have discussed a simple but powerful raster graphics package, SRGP. It lets the application program draw 2D primitives subject to various attributes that affect the appearance of those primitives. Drawing can be performed directly onto the screen canvas or onto an offscreen canvas of any desired size. Drawing can be restricted to a rectangular region of a canvas via the clip rectangle attribute. Besides the standard 2D shapes, SRGP also supports intra- and intercanvas copying of rectangular regions. Copying and drawing

Samsung/Google, EX1027, p. 83

can be affected by the write-mode attribute, allowing a destination pixel's current value to play a role in the determination of its new value.

SRGP also introduces the notion of logical input devices, which are high-level abstractions of physical input devices. The SRGP keyboard device abstracts the physical keyboard, and the locator device abstracts such devices as the mouse, the data tablet, and the joystick. Logical devices may operate either in sampled (polled) mode or in event mode. In event mode, a user action triggers the placing of an event report on the event queue, which the application may examine at its own convenience. In sample mode, the application continuously examines the device's measure for important changes.

Because SRGP scan converts primitives to their component pixels and does not store their original geometry, the only editing SRGP permits is the alteration of individual pixels, by drawing new primitives or by using the copyPixel operation on blocks of pixels. Object manipulations such as moving, deleting, or resizing must be done by the application program itself, which must respecify the updated image to SRGP.

Other systems offer a different set of features for graphics. For example, the PostScript language offers floating-point primitives and attributes, including far more general curved shapes and clipping facilities. PHIGS is a subroutine package that offers manipulation of hierarchically modeled objects, defined in a 3D floating-point world-coordinate system. These objects are stored in an editable database; the package automatically regenerates the image from this stored representation after any editing operation.

SRGP is a subroutine package, and many developers are finding that an interpreted language such as Adobe's PostScript provides maximal power and flexibility. Also, opinions differ on which should become standard—subroutine packages (integer or floating-point, with or without retention of primitives) or display languages such as PostScript that do not retain primitives. Each has its appropriate application domain, and we expect each to persist for some time.

In the next chapter, we see how SRGP does its drawing via scan conversion and clipping. In the following chapters, after an overview of hardware, we discuss the mathematics of transformations and 3D viewing in preparation for learning about PHIGS.

EXERCISES

2.1 SRGP runs in window environments, but does not allow the application to take advantage of multiple windows: The screen canvas is mapped to a single window on the screen, and no other canvases are visible. What changes would you make to the SRGP design and application-programmer interface to allow an application to take advantage of a window system?

2.2 An SRGP application can be fully machine-independent only if it uses solely the two colors 0 and 1. Develop a strategy for enhancing SRGP so that SRGP simulates color when necessary, allowing an application to be designed to take advantage of color but still to operate in a useful way on a bilevel display. Discuss the problems and conflicts that any such strategy creates.

2.3 Implement an animation sequence in which several trivial objects move and resize. First, generate each frame by erasing the screen and then specifying the objects in their new positions. Then, try *double-buffering*: use an offscreen canvas as a buffer into which each frame is drawn before being copied to the screen canvas. Compare the two methods' results. Also, consider the use of SRGP_copyPixel. Under what restricted circumstances is it useful for animation?

Samsung/Google, EX1027, p. 84

2.4 Implement a rubber-echo interaction, without using the built-in locator echo. Watch for artifacts, especially upon initiation and termination of the interaction feedback.

2.5 Implement nondestructive cursor tracking without using SRGP's built-in cursor echo. Use a bitmap or pixmap pattern to store a cursor's image, with 0s in the pattern representing transparency. Implement an **xor** cursor on a bilevel display, and a replace-mode cursor on a bilevel or color display. To test the tracking, you should perform a sampling loop with the SRGP locator device and move the cursor over a nonempty screen background.

2.6 Consider implementing the following feature in a painting application: The user can paint an **xor** swath that inverts the colors under the brush. It might seem that this is easily implemented by setting the write mode and then executing the code of Fig. 2.18. What complications arise? Propose solutions.

2.7 Some painting applications provide a "spray-painting" mode, in which passing the brush over an area affects an apparently random minority of the pixels in the area. Each time the brush passes over an area, different pixels are touched, so the more an area is touched by the brush, the "denser" the paint becomes. Implement a spray-painting interaction for a bilevel display. (Beware: The most obvious algorithms produce streaks or fail to provide increasing density. You will have to create a library of sparse bitmaps or patterns; see the reference manual for information on making custom patterns.)

2.8 Implement transparent-background text for bilevel displays, without using SRGP's built-in text primitive. Use an offscreen canvas to store the bitmap shape for each character, but support no more than six characters—this is not a lesson in font design! (Hint: You may have to use two different algorithms to support both colors 0 and 1.)

2.9 A drawing program can update the screen after a deletion operation by filling the deleted object's shape with the application screen background pattern. This of course may damage other objects on the screen. Why is it not sufficient to repair the damage by simply respecifying all objects whose rectangular extents intersect the extent of the deleted object? Discuss solutions to the problem of optimizing damage repair.

2.10 Implement a procedure that draws text centered within an opaque rectangle with a thin border. Allow the caller to specify the colors for the text, background, and border; the screen position at which the center of the "button" should be placed; a pair of min/max dimensions for both width and height; and the font and the text string itself. If the string cannot fit on one line within the button at its maximum length, break the string at appropriate places (e.g., spaces) to make multiline text for the button.

2.11 Implement an onscreen valuator logical input device that allows the user to specify a temperature by using the mouse to vary the length of a simulated column of mercury. The device's attributes should include the range of the measure, the initial measure, the desired granularity of the measure (e.g., accuracy to 2 F degrees), and the desired length and position of the thermometer's screen image. To test your device, use an interaction that simulates an indefinite waitEvent where the only active device is the valuator.

2.12 Imagine customizing an SRGP implementation by adding an onscreen valuator device (like that described in Exercise 2.11) to the input model and supporting it for both event and sample modes. What kinds of problems might arise if the implementation is installed on a workstation having only one physical locator device? Propose solutions.

2.13 Implement a "rounded-rectangle"primitive—a rectangle whose corners are rounded, each corner being an ellipse arc of 90 rectangular degrees. Allow the application to have control of the radii of the ellipse arc. Support both outlined and filled versions.

Samsung/Google, EX1027, p. 85

PROGRAMMING PROJECTS

2.14 Implement the pull-down menu package whose high-level design is presented in code fragments in Sections 2.2.6, 2.3.1, and 2.3.3. Have the package initialize the menu bar and menu bodies by reading strings from an input file. Allow the program to deactivate a menu to make the header disappear, and to activate a menu (with its horizontal position on the menu bar as a parameter) to make that menu appear.

2.15 Enhance your menu package from Exercise 2.14 by implementing disabling of selected menu items. Disabled items in a menu body should appear "grayed out"; since SRGP does not support the drawing of text using a pen style, on a bilevel display you must paint over solid text using a write mode in order to achieve this effect.

2.16 Enhance your menu package from Exercise 2.14 by highlighting the item to which the locator currently points while the user is choosing an item from the menu body.

2.17 Implement a layout application that allows the user to place objects in a square subregion of the screen. Ellipses, rectangles, and equilateral triangles should be supported. The user will click on a screen button to select an object type or to initiate an action (redraw screen, save scene to file, restore scene from file, or quit).

2.18 Add object editing to your layout application from Exercise 2.17. The user must be able to delete, move, and resize or rescale objects. Use this simple pick-correlation method: scan the objects in the application database and choose the first object whose rectangular extent encloses the locator position. (Show that this naive method has a disturbing side effect: It is possible for a visible object to be unpickable!) Be sure to give the user feedback by highlighting the currently selected object.

2.19 Add an extra half dimension to your layout application from Exercise 2.17 by implementing overlap priority. The user must be able to push/pop an object (force its priority to be the very lowest/highest). Enhance pick correlation to use overlap priority to resolve conflicts. How does the push/pop functionality, along with the use of priority by the pick correlator, allow the user to override the inaccuracy of naive pick correlation?

2.20 Optimize the screen-update algorithm of your layout application from Exercise 2.17 using the results of Exercise 2.9, so that a minimum number of objects is respecified in response to an edit operation.

2.21 Enhance your layout application from Exercise 2.17 so that the keyboard and locator are enabled simultaneously, to provide keyboard abbreviations for common operations. For example, pressing the "d" key could delete the currently selected object.

2.22 Design and implement analytical techniques for pick correlation for the three types of objects supported by your layout application from Exercise 2.17. Your new techniques should provide full accuracy; the user should no longer have to use pop/push to pick a visible low-priority object.

Samsung/Google, EX1027, p. 86

3 Basic Raster Graphics Algorithms for Drawing 2D Primitives

A raster graphics package approximates mathematical ("ideal") primitives, described in terms of vertices on a Cartesian grid, by sets of pixels of the appropriate intensity of gray or color. These pixels are stored as a bitmap or pixmap in CPU memory or in a frame buffer. In the previous chapter, we studied the features of SRGP, a typical raster graphics package, from an *application programmer's* point of view. The purpose of this chapter is to look at SRGP from a *package implementor's* point of view—that is, in terms of the fundamental algorithms for scan converting primitives to pixels, subject to their attributes, and for clipping them against an upright clip rectangle. Examples of scan-converted and clipped primitives are shown in Fig. 3.1.

More advanced algorithms that handle features not supported in SRGP are used in more sophisticated and complex packages; such algorithms are treated in Chapter 19. The algorithms in this chapter are discussed in terms of the 2D integer Cartesian grid, but most of the scan-conversion algorithms can be extended to floating point, and the clipping algorithms can be extended both to floating point and to 3D. The final section introduces the concept of antialiasing—that is, minimizing jaggies by making use of a system's ability to vary a pixel's intensity.

3.1 OVERVIEW

3.1.1 Implications of Display-System Architecture

The fundamental conceptual model of Section 1.7 presents a graphics package as the system that mediates between the application program (and its application data structure/model) and the display hardware. The package gives the application program a device-

Samsung/Google, EX1027, p. 87

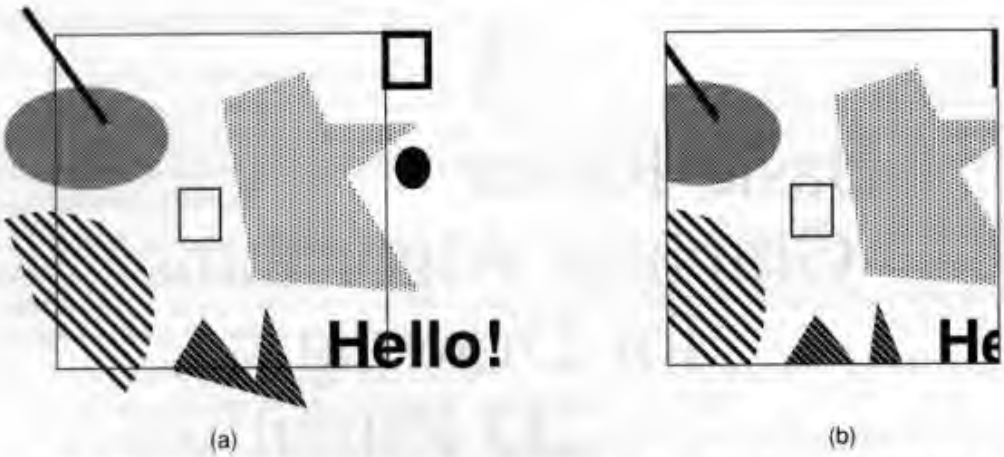


Fig. 3.1 Clipping SRGP primitives to a rectangular clip region. (a) Primitives and clipping rectangle. (b) Clipped results.

independent interface to the hardware, as shown in Fig. 3.2, where SRGP's procedures are partitioned into those forming an output pipeline and those forming an input pipeline.

In the *output pipeline*, the application program takes descriptions of objects in terms of primitives and attributes stored in or derived from an application model or data structure, and specifies them to the graphics package, which in turn clips and scan converts them to the pixels seen on the screen. The package's primitive-generation procedures specify *what* is to be generated, the attribute procedures specify *how* primitives are to be generated, the SRGP_copyPixel procedure specifies *how* images are to be modified, and the canvas-control procedures specify *where* the images are to be generated. In the *input pipeline*, a user interaction at the display end is converted to measure values returned by the package's sampling or event-driven input procedures to the application program; it typically uses those

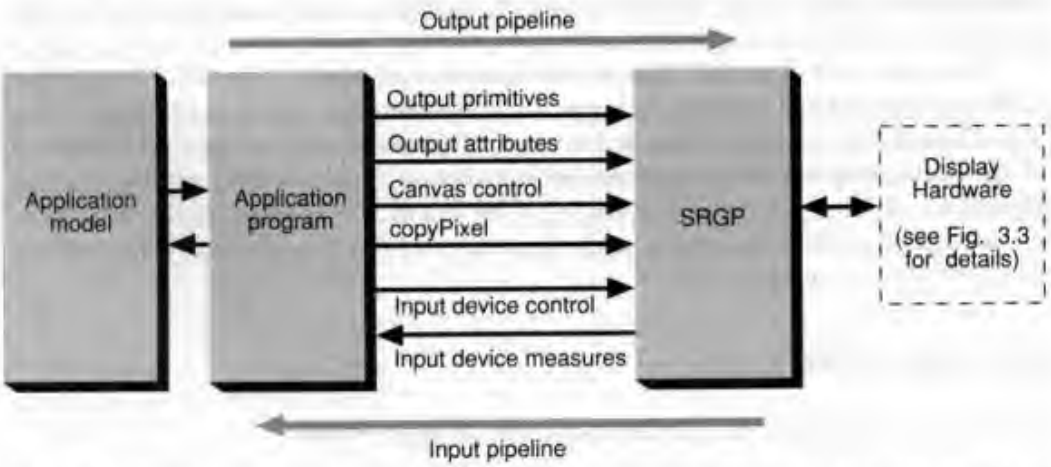


Fig. 3.2 SRGP as intermediary between the application program and the graphics system, providing output and input pipelines.

Samsung/Google, EX1027, p. 88

values to modify the model or the image on the screen. Procedures relating to input include those to initialize and control input devices and those to obtain the latter's measures during interaction. We do not cover either SRGP's canvas management or its input handling in this book, since these topics have little to do with raster graphics and are primarily data-structure and low-level systems-software issues, respectively.

An SRGP implementation must communicate with a potentially wide variety of display devices. Some display systems are attached as peripherals with their own internal frame buffers and display controllers. These display controllers are processors specialized to interpret and execute drawing commands that generate pixels into the frame buffer. Other, simpler systems are refreshed directly from the memory used by the CPU. Output-only subsets of the package may drive raster hardcopy devices. These various types of hardware architectures are discussed in more detail in Chapters 4 and 18. In any display-system architecture, the CPU must be able to read and write individual pixels in the frame buffer. It is also convenient to be able to move rectangular blocks of pixels to and from the frame buffer to implement the copyPixel (bitBlt) type of operation. This facility is used not for generating primitives directly but to make portions of offscreen bitmaps or pixmaps visible and to save and restore pieces of the screen for window management, menu handling, scrolling, and so on.

Whereas all implementations for systems that refresh from CPU memory are essentially identical because all the work is done in software, implementations for display controller and hardcopy systems vary considerably, depending on what the respective hardware devices can do by themselves and what remains for the software to do. Naturally, in any architecture, software scan conversion must be used to generate both primitives and attributes not directly supported in hardware. Let's look briefly at the range of architectures and implementations.

Displays with frame buffers and display controllers. SRGP has the least amount of work to do if it drives a display controller that does its own scan conversion and handles all of SRGP's primitives and attributes directly. In this case, SRGP needs only to convert its internal representation of primitives, attributes, and write modes to the formats accepted by the display peripheral that actually draws the primitives (Fig. 3.3 a).

The display-controller architecture is most powerful when memory mapping allows the CPU to access the frame buffer directly and the display controller to access the CPU's memory. The CPU can then read and write individual pixels and copyPixel blocks of pixels with normal CPU instructions, and the display controller can scan convert into offscreen canvases and also use its copyPixel instruction to move pixels between the two memories or within its own frame buffer. When the CPU and the display controller can run asynchronously, there must be synchronization to avoid memory conflicts. Often, the display controller is controlled by the CPU as a coprocessor. If the display peripheral's display controller can only scan convert into its own frame buffer and cannot write pixels into CPU memory, we need a way to generate primitives in an offscreen canvas. The package then uses the display controller for scan conversion into the screen canvas but must do its own software scan conversion for offscreen canvases. The package can, of course, copyPixel images scan converted by the hardware from the frame buffer to offscreen canvases.

Samsung/Google, EX1027, p. 89

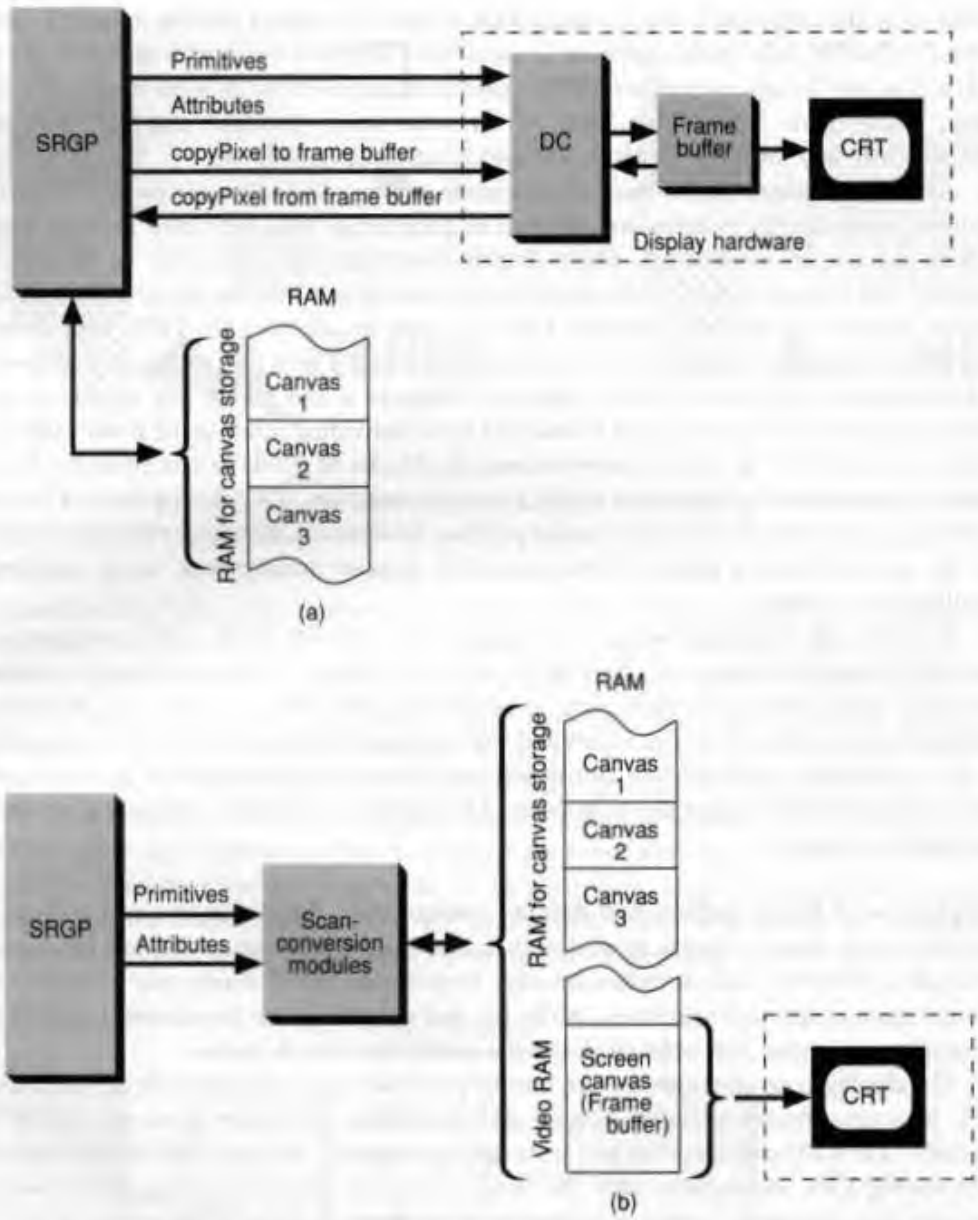


Fig. 3.3 SRGP driving two types of display systems. (a) Display peripheral with display controller and frame buffer. (b) No display controller, memory-shared frame buffer.

Displays with frame buffers only. For displays without a display controller, SRGP does its own scan conversion into both offscreen canvases and the frame buffer. A typical organization for such an SRGP implementation that drives a shared-memory frame buffer is shown in Fig. 3.3 (b). Note that we show only the parts of memory that constitute the frame

Samsung/Google, EX1027, p. 90

buffer and store the canvases managed by SRGP; the rest of the memory is occupied by all the usual software and data, including, of course, SRGP itself.

Hardcopy devices. As explained in Chapter 4, hardcopy devices range in their capabilities along the same spectrum as display systems. The simplest devices accept only one scan line at a time and rely on the software to provide that scan line exactly when it is to be imaged on film or on paper. For such simple hardware, SRGP must generate a complete bitmap or pixmap and scan it out one line at a time to the output device. Slightly smarter devices can accept an entire frame (page) at a time. Yet more powerful equipment has built-in scan-conversion hardware, often called raster image processors (RIPs). At the high end of the scale, PostScript printers have internal "engines" that read PostScript programs describing pages in a device-independent fashion; they interpret such programs to produce the primitives and attributes that are then scan converted. The fundamental clipping and scan-conversion algorithms are essentially independent of the raster device's output technology; therefore, we need not address hardcopy devices further in this chapter.

3.1.2 The Output Pipeline in Software

Here we examine the output pipeline driving simple frame-buffer displays only in order to address the problems of software clipping and scan conversion. The various algorithms introduced are discussed at a general, machine-independent level, so they apply to both software and hardware (or microcode) implementations.

As each output primitive is encountered by SRGP, the package *scan converts* the primitive: Pixels are written in the current canvas according to their applicable attributes and current write mode. The primitive is also *clipped* to the clip rectangle; that is, pixels belonging to the primitive that are outside the clip region are not displayed. There are several ways of doing clipping. The obvious technique is to clip a primitive prior to scan conversion by computing its analytical intersections with the clip-rectangle boundaries; these intersection points are then used to define new vertices for the clipped version of the primitive. The advantage of clipping before scan converting is, of course, that the scan converter must deal with only the clipped version of the primitive, not with the original (possibly much larger) one. This technique is used most often for clipping lines, rectangles, and polygons, for which clipping algorithms are fairly simple and efficient.

The simplest, brute-force clipping technique, called *scissoring*, is to scan convert the entire primitive but to write only the visible pixels in the clip-rectangle region of the canvas. In principle, this is done by checking each pixel's coordinates against the (x, y) bounds of the rectangle before writing that pixel. In practice, there are shortcuts that obviate having to check adjacent pixels on a scan line, as we shall see later. This type of clipping is thus accomplished on the fly; if the bounds check can be done quickly (e.g., by a tight inner loop running completely in microcode or in an instruction cache), this approach may actually be faster than first clipping the primitive and then scan converting the resulting, clipped portions. It also generalizes to arbitrary clip regions.

A third technique is to generate the entire collection of primitives into a temporary canvas and then to copyPixel only the contents of the clip rectangle to the destination canvas. This approach is wasteful of both space and time, but is easy to implement and is often used for text. Data structures for minimizing this overhead are discussed in Chapter 19.

Samsung/Google, EX1027, p. 91

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

3.2 SCAN CONVERTING LINES

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

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

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

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

Samsung/Google, EX1027, p. 92

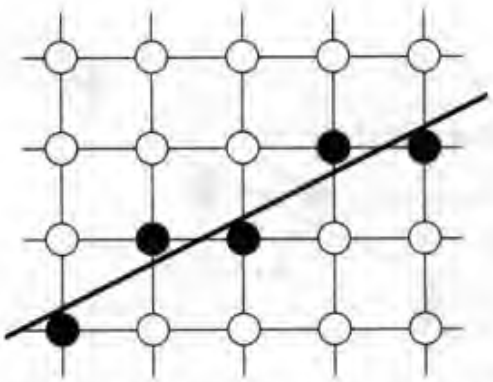


Fig. 3.4 A scan-converted line showing intensified pixels as black circles.

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

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

3.2.1 The Basic Incremental Algorithm

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

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

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

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

[1] In Chapter 19, we discuss various measures of closeness for lines and general curves (also called *error measures*).

Samsung/Google, EX1027, p. 93

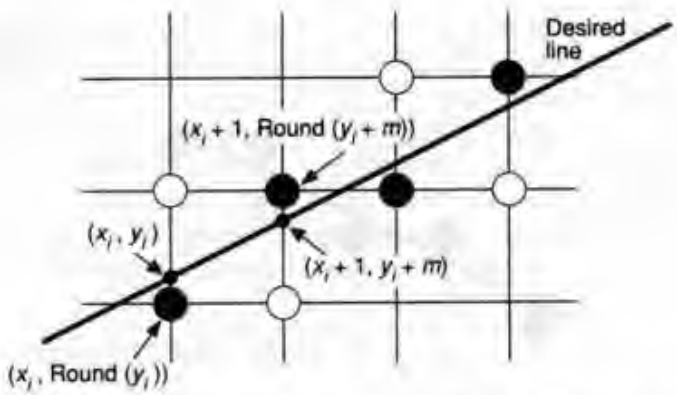


Fig. 3.5 Incremental calculation of (x_i, y_i).

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

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

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

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

3.2.2 Midpoint Line Algorithm

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

[2] If such a low-level procedure is not available, the SRGP_pointCoord procedure may be used, as described in the SRGP reference manual.

Samsung/Google, EX1027, p. 94

```
void Line (                                  /* Assumes -1 ≤ m ≤ 1, x0 < x1 */
    int x0, int y0,                          /* Left endpoint */
    int x1, int y1,                          /* Right endpoint */
    int value)                               /* Value to place in line's pixels */
{
    int x;                                   /* x runs from x0 to x1 in unit increments. */

    double dy = y1 - y0;
    double dx = x1 - x0;
    double m = dy / dx;
    double y = y0;
    for (x = x0; x <= x1; x++) {
        WritePixel (x, Round (y), value);    /* Set pixel to value */
        y += m;                              /* Step y by slope m */
    }
} /* Line */
```

Fig. 3.6 The incremental line scan-conversion algorithm.

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

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

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

Samsung/Google, EX1027, p. 95

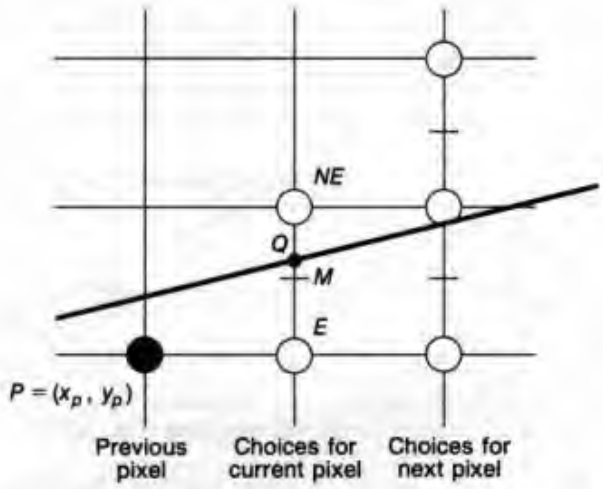


Fig. 3.7 The pixel grid for the midpoint line algorithm, showing the midpoint *M*, and the *E* and *NE* pixels to choose between.

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

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

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

therefore,

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

Here $a = dy$, $b = -dx$, and $c = B \cdot dx$ in the implicit form.[4]

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

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

[3]This functional form extends nicely to the implicit formulation of both circles and ellipses.

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

Samsung/Google, EX1027, p. 96

incremented by one step in the x direction. Then,

$$d_{new} = F(x_P + 2, y_P + \tfrac{1}{2}) = a(x_P + 2) + b(y_P + \tfrac{1}{2}) + c,$$

but

$$d_{old} = a(x_P + 1) + b(y_P + \tfrac{1}{2}) + c.$$

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

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

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

$$d_{new} = F(x_P + 2, y_P + \tfrac{3}{2}) = a(x_P + 2) + b(y_P + \tfrac{3}{2}) + c.$$

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

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

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

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

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

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

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

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

Samsung/Google, EX1027, p. 97

```
void MidpointLine (int x0, int y0, int x1, int y1, int value)
{
    int dx = x1 - x0;
    int dy = y1 - y0;
    int d = 2 * dy - dx;            /* Initial value of d */
    int incrE = 2 * dy;             /* Increment used for move to E */
    int incrNE = 2 * (dy - dx);     /* Increment used for move to NE */
    int x = x0;
    int y = y0;
    WritePixel (x, y, value);       /* The start pixel */

    while (x < x1) {
        if (d <= 0) {               /* Choose E */
            d += incrE;
            x++;
        } else {                    /* Choose NE */
            d += incrNE;
            x++;
            y++;
        }
        WritePixel (x, y, value);   /* The selected pixel closest to the line */
    } /* while */

} /* MidpointLine */
```

Fig. 3.8 The midpoint line scan-conversion algorithm.

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

3.2.3 Additional Issues

Endpoint order. Among the complications to consider is that we must ensure that a line from P_0 to P_1 contains the same set of pixels as the line from P_1 to P_0, so that the appearance of the line is independent of the order of specification of the endpoints. The only place where the choice of pixel is dependent on the direction of the line is where the line passes exactly through the midpoint and the decision variable is zero; going left to right, we chose to pick E for this case. By symmetry, while going from right to left, we would also expect to choose W for $d = 0$, but that would choose a pixel one unit up in y relative to the one chosen for the left-to-right scan. We therefore need to choose SW when $d = 0$ for right-to-left scanning. Similar adjustments need to be made for lines at other slopes.

Samsung/Google, EX1027, p. 98

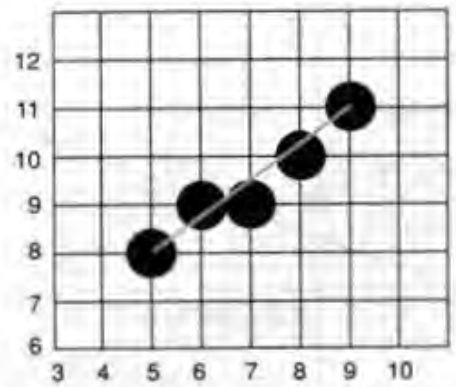


Fig. 3.9 The midpoint line from point (5, 8) to point (9, 11).

The alternative solution of switching a given line's endpoints as needed so that scan conversion always proceeds in the same direction does not work when we use line styles. The line style always "anchors" the specified write mask at the start point, which would be the bottom-left point, independent of line direction. That does not necessarily produce the desired visual effect. In particular, for a dot-dash line pattern of, say, 111100, we would like to have the pattern start at whichever start point is specified, not automatically at the bottom-left point. Also, if the algorithm always put endpoints in a canonical order, the pattern might go left to right for one segment and right to left for the adjoining segment, as a function of the second line's slope; this would create an unexpected discontinuity at the shared vertex, where the pattern should follow seamlessly from one line segment to the next.

Starting at the edge of a clip rectangle. Another issue is that we must modify our algorithm to accept a line that has been analytically clipped by one of the algorithms in Section 3.12. Fig. 3.10(a) shows a line being clipped at the left edge, $x = x_{min}$, of the clip rectangle. The intersection point of the line with the edge has an integer x coordinate but a real y coordinate. The pixel at the left edge, $(x_{min}, \text{Round}(mx_{min} + B))$, is the same pixel that would be drawn at this x value for the unclipped line by the incremental algorithm.[5] Given this initial pixel value, we must next initialize the decision variable at the midpoint between the E and NE positions in the next column over. It is important to realize that this strategy produces the correct sequence of pixels, while clipping the line at the x_{min} boundary and then scan converting the clipped line from $(x_{min}, \text{Round}(mx_{min} + B))$ to (x_1, y_1) using the integer midpoint line algorithm would not—that clipped line has a different slope!

The situation is more complicated if the line intersects a horizontal rather than a vertical edge, as shown in Fig. 3.10 (b). For the type of shallow line shown, there will be multiple pixels lying on the scan line $y = y_{min}$ that correspond to the bottom edge of the clip region. We want to count each of these as inside the clip region, but simply computing the analytical intersection of the line with the $y = y_{min}$ scan line and then rounding the x value of the intersection point would produce pixel A, not the leftmost point of the span of pixels shown, pixel B. From the figure, it is clear that the leftmost pixel of the span, B, is the one

[5] When $mx_{min} + B$ lies exactly halfway between horizontal grid lines, we actually must round down. This is a consequence of choosing pixel E when $d = 0$.

Samsung/Google, EX1027, p. 99

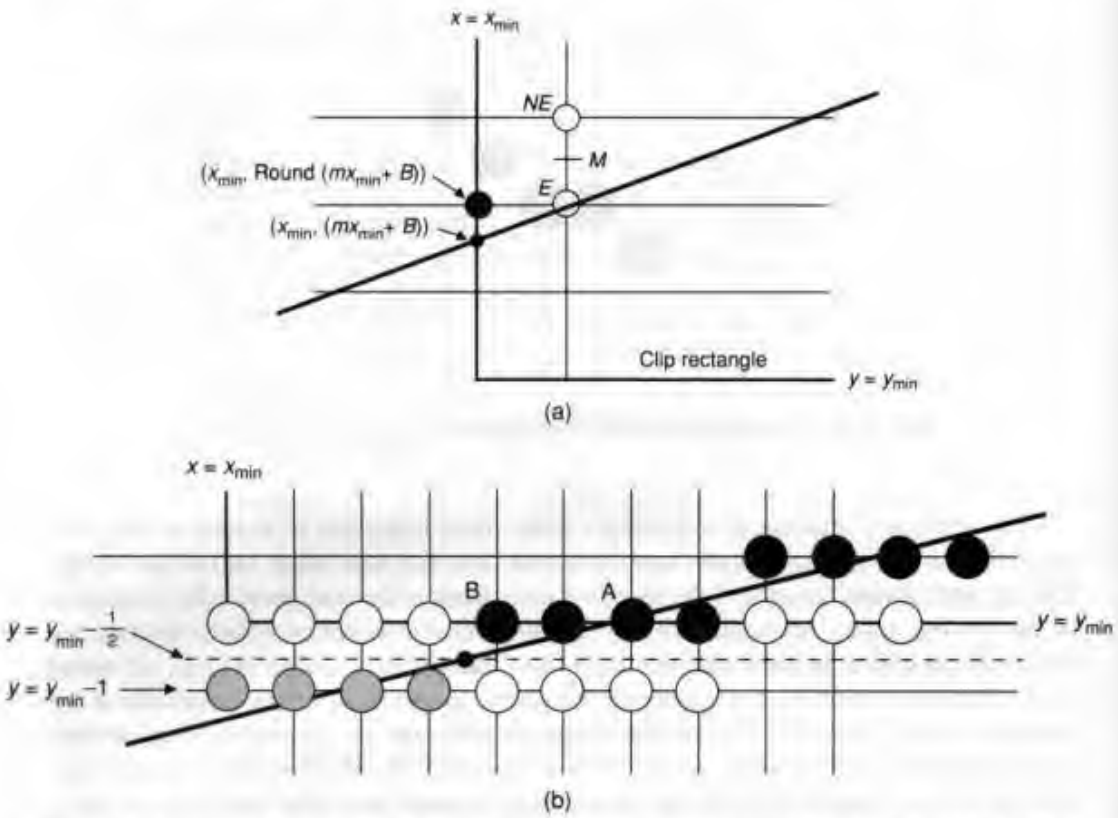


Fig. 3.10 Starting the line at a clip boundary. (a) Intersection with a vertical edge. (b) Intersection with a horizontal edge (gray pixels are on the line but are outside the clip rectangle).

that lies just above and to the right of the place on the grid where the line first crosses above the midpoint $y = y_{min} - \frac{1}{2}$. Therefore, we simply find the intersection of the line with the horizontal line $y = y_{min} - \frac{1}{2}$, and round up the x value; the first pixel, B, is then the one at $(\text{Round}(x_{y_{min}-\frac{1}{2}}), y_{min})$.

Finally, the incremental midpoint algorithm works even if endpoints are specified in a floating-point raster graphics package; the only difference is that the increments are now reals, and the arithmetic is done with reals.

Varying the intensity of a line as a function of slope. Consider the two scan converted lines in Fig. 3.11. Line B, the diagonal line, has a slope of 1 and hence is $\sqrt{2}$ times as long as A, the horizontal line. Yet the same number of pixels (10) is drawn to represent each line. If the intensity of each pixel is I, then the intensity per unit length of line A is I, whereas for line B it is only $I/\sqrt{2}$; this discrepancy is easily detected by the viewer. On a bilevel display, there is no cure for this problem, but on an n-bits-per-pixel system we can compensate by setting the intensity to be a function of the line's slope. Antialiasing, discussed in Section 3.17, achieves an even better result by treating the line as a thin rectangle and computing

Samsung/Google, EX1027, p. 100

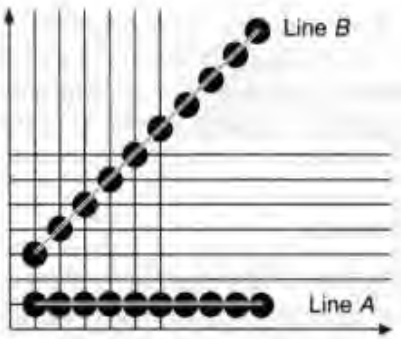


Fig. 3.11 Varying intensity of raster lines as a function of slope.

appropriate intensities for the multiple pixels in each column that lie in or near the rectangle.

Treating the line as a rectangle is also a way to create thick lines. In Section 3.9, we show how to modify the basic scan-conversion algorithms to deal with thick primitives and with primitives whose appearance is affected by line-style and pen-style attributes. Chapter 19 treats several other enhancements of the fundamental algorithms, such as handling endpoint shapes and creating joins between lines with multiple-pixel width.

Outline primitives composed of lines. Knowing how to scan convert lines, how do we scan convert primitives made from lines? Polylines can be scan-converted one line segment at a time. Scan converting rectangles and polygons as area-defining primitives could be done a line segment at a time but that would result in some pixels being drawn that lie outside a primitive's area—see Sections 3.5 and 3.6 for special algorithms to handle this problem. Care must be taken to draw shared vertices of polylines only once, since drawing a vertex twice causes it to change color or to be set to background when writing in **xor** mode to a screen, or to be written at double intensity on a film recorder. In fact, other pixels may be shared by two line segments that lie close together or cross as well. See Section 19.7 and Exercise 3.8 for a discussion of this, and of the difference between a polyline and a sequence of connected line segments.

3.3 SCAN CONVERTING CIRCLES

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

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

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

Samsung/Google, EX1027, p. 101

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

3.3.1 Eight-Way Symmetry

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

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

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

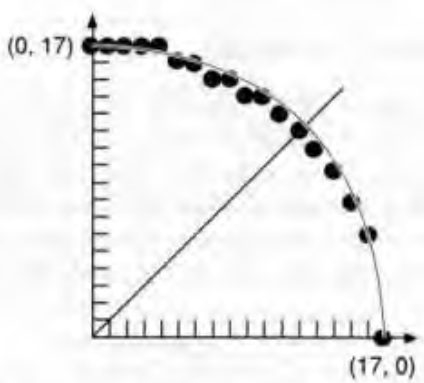


Fig. 3.12 A quarter circle generated with unit steps in x, and with y calculated and then rounded. Unique values of y for each x produce gaps.

Samsung/Google, EX1027, p. 102

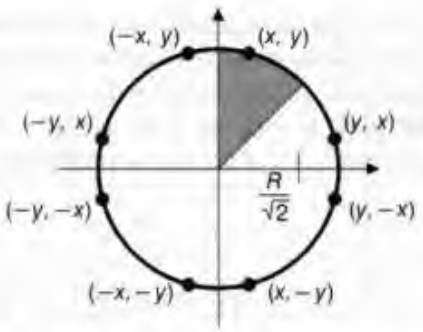


Fig. 3.13 Eight symmetrical points on a circle.

3.3.2 Midpoint Circle Algorithm

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

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

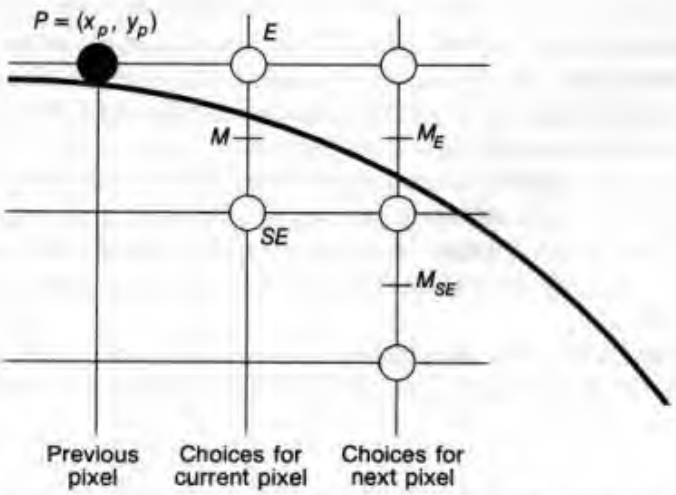


Fig. 3.14 The pixel grid for the midpoint circle algorithm showing M and the pixels E and SE to choose between.

Samsung/Google, EX1027, p. 103

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

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

$$d_{old} = F(x_P + 1, y_P - \tfrac{1}{2}) = (x_P + 1)^2 + (y_P - \tfrac{1}{2})^2 - R^2.$$

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

$$d_{new} = F(x_P + 2, y_P - \tfrac{1}{2}) = (x_P + 2)^2 + (y_P - \tfrac{1}{2})^2 - R^2,$$

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

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

$$d_{new} = F(x_P + 2, y_P - \tfrac{3}{2}) = (x_P + 2)^2 + (y_P - \tfrac{3}{2})^2 - R^2.$$

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

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

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

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

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

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

---

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

Samsung/Google, EX1027, p. 104

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

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

Fig. 3.15 The midpoint circle scan-conversion algorithm.

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

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

Second-order differences. We can improve the performance of the midpoint circle algorithm by using the incremental computation technique even more extensively. We noted that the Δ functions are linear equations, and we computed them directly. Any polynomial can be computed incrementally, however, as we did with the decision variables for both the line and the circle. In effect, we are calculating *first-* and *second-order partial differences*, a useful technique that we encounter again in Chapters 11 and 19. The strategy is to evaluate the function directly at two adjacent points, to calculate the difference (which, for polynomials, is always a polynomial of lower degree), and to apply that difference in each iteration.

If we choose E in the current iteration, the point of evaluation moves from (x_P, y_P) to $(x_P + 1, y_P)$. As we saw, the first-order difference is $\Delta_{E_{old}}$ at $(x_P, y_P) = 2x_P + 3$. Therefore,

$$\Delta_{E_{new}} \text{ at } (x_P + 1, y_P) = 2(x_P + 1) + 3,$$

and the second-order difference is $\Delta_{E_{new}} - \Delta_{E_{old}} = 2$.

Samsung/Google, EX1027, p. 105

```
void MidpointCircle (int radius, int value)
/* Assumes center of circle is at origin. Integer arithmetic only */
{
    int x = 0;
    int y = radius;
    int d = 1 - radius;
    CirclePoints (x, y, value);

    while (y > x) {
        if (d < 0)          /* Select E */
            d += 2 * x + 3;
        else {              /* Select SE */
            d += 2 * (x - y) + 5;
            y--;
        }
        x++;
        CirclePoints (x, y, value);
    } /* while */
} /* MidpointCircle */
```

Fig. 3.16 The integer midpoint circle scan-conversion algorithm.

Similarly, $\Delta_{SE_{old}}$ at $(x_P, y_P) = 2x_P - 2y_P + 5$. Therefore,

$$\Delta_{SE_{new}} \text{ at } (x_P + 1, y_P) = 2(x_P + 1) - 2y_P + 5,$$

and the second-order difference is $\Delta_{SE_{new}} - \Delta_{SE_{old}} = 2$.

If we choose SE in the current iteration, the point of evaluation moves from (x_P, y_P) to $(x_P + 1, y_P - 1)$. Therefore,

$$\Delta_{E_{new}} \text{ at } (x_P + 1, y_P - 1) = 2(x_P + 1) + 3,$$

and the second-order difference is $\Delta_{E_{new}} - \Delta_{E_{old}} = 2$. Also,

$$\Delta_{SE_{new}} \text{ at } (x_P + 1, y_P - 1) = 2(x_P + 1) - 2(y_P - 1) + 5,$$

and the second-order difference is $\Delta_{SE_{new}} - \Delta_{SE_{old}} = 4$.

The revised algorithm then consists of the following steps: (1) choose the pixel based on the sign of the variable d computed during the previous iteration; (2) update the decision variable d with either Δ_E or Δ_{SE}, using the value of the corresponding Δ computed during the previous iteration; (3) update the Δs to take into account the move to the new pixel, using the constant differences computed previously; and (4) do the move. Δ_E and Δ_{SE} are initialized using the start pixel $(0, R)$. The revised procedure using this technique is shown in Fig. 3.18.

Samsung/Google, EX1027, p. 106

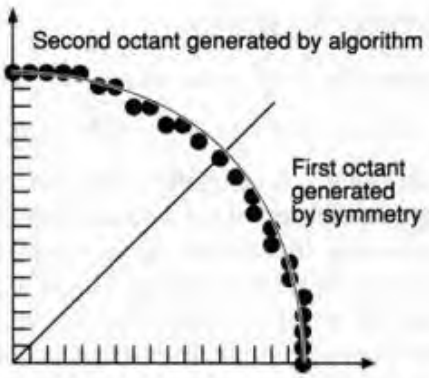


Fig. 3.17 Second octant of circle generated with midpoint algorithm, and first octant generated by symmetry.

```
void MidpointCircle (int radius, int value)
/* This procedure uses second-order partial differences to compute increments */
/* in the decision variable. Assumes center of circle is at origin */
{
    int x = 0;
    int y = radius;
    int d = 1 - radius;
    int deltaE = 3;
    int deltaSE = -2 * radius + 5;
    CirclePoints (x, y, value);

    while (y > x) {
        if (d < 0) {                /* Select E */
            d += deltaE;
            deltaE += 2;
            deltaSE += 2;
        } else {
            d += deltaSE;           /* Select SE */
            deltaE += 2;
            deltaSE += 4;
            y--;
        }
        x++;
        CirclePoints (x, y, value);
    }  /* while */
}  /* MidpointCircle */
```

Fig. 3.18 Midpoint circle scan-conversion algorithm using second-order differences.

Samsung/Google, EX1027, p. 107

3.4 SCAN CONVERTING ELLIPSES

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

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

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

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

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

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

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

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

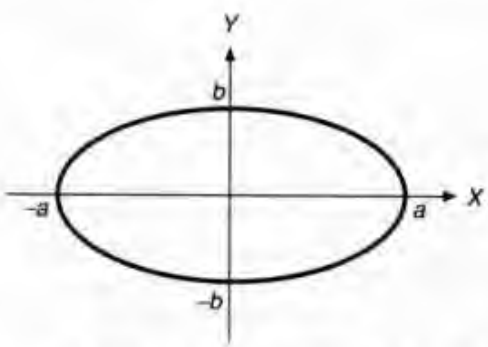


Fig. 3.19 Standard ellipse centered at the origin.

Samsung/Google, EX1027, p. 108

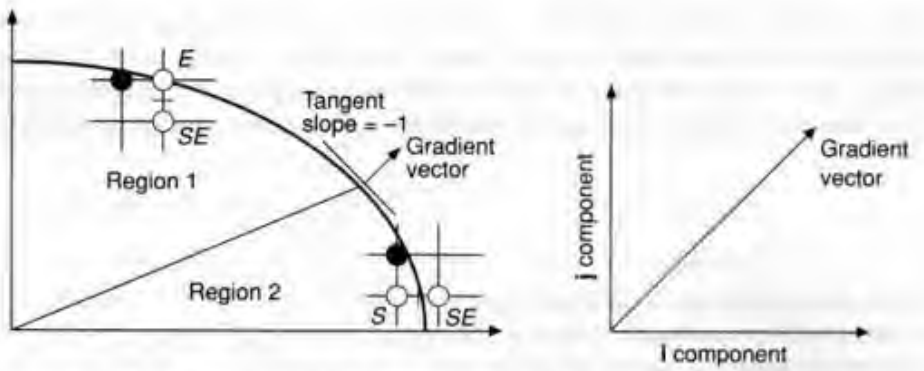


Fig. 3.20 Two regions of the ellipse defined by the 45° tangent.

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

$$d_{old} = F(x_P + 1, y_P - \tfrac{1}{2}) = b^2(x_P + 1)^2 + a^2(y_P - \tfrac{1}{2})^2 - a^2b^2,$$

$$d_{new} = F(x_P + 2, y_P - \tfrac{1}{2}) = b^2(x_P + 2)^2 + a^2(y_P - \tfrac{1}{2})^2 - a^2b^2.$$

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

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

$$d_{new} = F(x_P + 2, y_P - \tfrac{3}{2}) = b^2(x_P + 2)^2 + a^2(y_P - \tfrac{3}{2})^2 - a^2b^2.$$

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

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

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

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

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

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

Samsung/Google, EX1027, p. 109

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

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

    int x = 0;
    int y = b;
    double d1 = b^2 - (a^2 b) + (0.25 a^2);
    EllipsePoints (x, y, value);              /* The 4-way symmetrical WritePixel */

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

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

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

Samsung/Google, EX1027, p. 110

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

3.5 FILLING RECTANGLES

The task of filling primitives can be broken into two parts: the decision of which pixels to fill (this depends on the shape of the primitive, as modified by clipping), and the easier decision of with what value to fill them. We first discuss filling unclipped primitives with a solid color; we deal with pattern filling in Section 3.8. In general, determining which pixels to fill consists of taking successive scan lines that intersect the primitive and filling in *spans* of adjacent pixels that lie inside the primitive from left to right.

To fill a rectangle with a solid color, we set each pixel lying on a scan line running from the left edge to the right edge to the same pixel value; i.e., fill each span from x_{min} to x_{max}. Spans exploit a primitive's *spatial coherence*: the fact that primitives often do not change from pixel to pixel within a span or from scan line to scan line. We exploit coherence in general by looking for only those pixels at which changes occur. For a solidly shaded primitive, all pixels on a span are set to the same value, which provides *span coherence*. The solidly shaded rectangle also exhibits strong *scan-line coherence* in that consecutive scan lines that intersect the rectangle are identical; later, we also use *edge coherence* for the edges of general polygons. We take advantage of various types of coherence not only for scan converting 2D primitives, but also for rendering 3D primitives, as discussed in Section 15.2.

Being able to treat multiple pixels in a span identically is especially important because we should write the frame buffer one word at a time to minimize the number of time-consuming memory accesses. For a bilevel display, we thus write 16 or 32 pixels at a time; if spans are not word-aligned, the algorithm must do suitable masking of words containing fewer than the full set of pixels. The need for writing memory efficiently is entirely similar for implementing copyPixel, as briefly discussed in Section 3.16. In our code, we concentrate on defining spans and ignore the issue of writing memory efficiently; see Chapters 4 and 19 and Exercise 3.13.

Rectangle scan conversion is thus simply a nested **for** loop:

```
for (y from ymin to ymax of the rectangle)      /* By scan line */
    for (x from xmin to xmax )                  /* By pixel in span */
        WritePixel (x, y, value);
```

An interesting problem arises in this straightforward solution, similar to the problem of scan converting a polyline with line segments that share pixels. Consider two rectangles that share a common edge. If we scan convert each rectangle in turn, we will write the pixels on the shared edge twice, which is undesirable, as noted earlier. This problem is a manifestation of a larger problem of area-defining primitives, that of defining which pixels belong to a primitive and which pixels do not. Clearly, those pixels that lie in the mathematical interior of an area-defining primitive belong to that primitive. But what about those pixels on the boundary? If we were looking at a single rectangle (or just thinking

Samsung/Google, EX1027, p. 111

about the problem in a mathematical way), a straightforward answer would be to include the pixels on the boundary, but since we want to avoid the problem of scan converting shared edges twice, we must define some rule that assigns boundary pixels uniquely.

A simple rule is to say that a boundary pixel—that is, a pixel lying on an edge—is not considered part of the primitive if the halfplane defined by that edge and containing the primitive lies below or to the left of the edge. Thus, pixels on left and bottom edges will be drawn, but pixels that lie on top and right edges will not be drawn. A shared vertical edge therefore "belongs" to the rightmost of the two sharing rectangles. In effect, spans within a rectangle represent an interval that is closed on the left end and open on the right end.

A number of points must be made about this rule. First, it applies to arbitrary polygons as well as to rectangles. Second, the bottom-left vertex of a rectangle still would be drawn twice—we need another rule to deal with that special case, as discussed in the next section. Third, we may apply the rule also to unfilled rectangles and polygons. Fourth, the rule causes each span to be missing its rightmost pixel, and each rectangle to be missing its topmost row. These problems illustrate that there is no "perfect" solution to the problem of not writing pixels on (potentially) shared edges twice, but implementors generally consider that it is better (visually less distracting) to have missing pixels at the right and top edge than it is to have pixels that disappear or are set to unexpected colors in **xor** mode.

3.6 FILLING POLYGONS

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

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

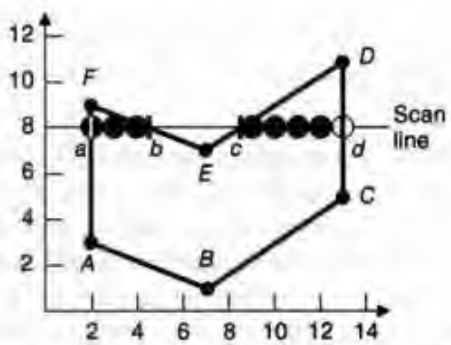


Fig. 3.22 Polygon and scan line 8.

Samsung/Google, EX1027, p. 112

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

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

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

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

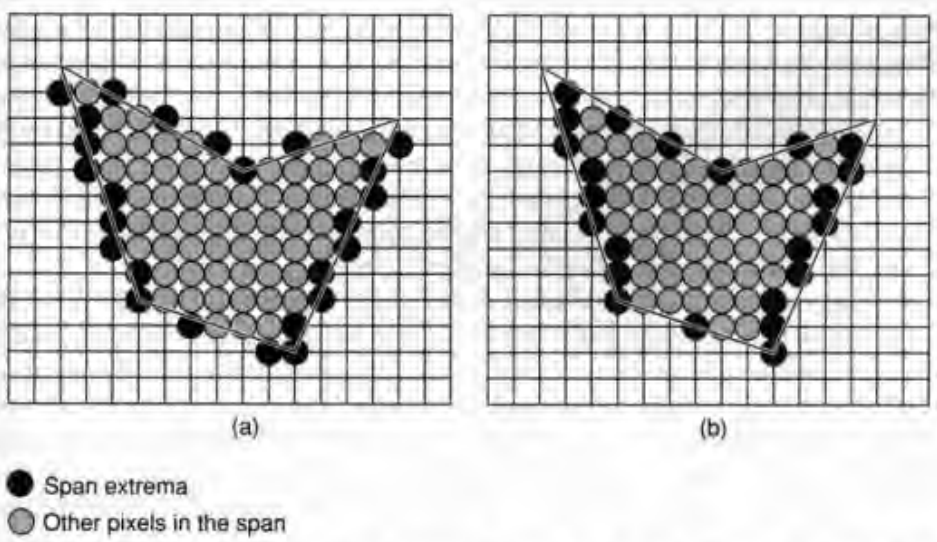


Fig. 3.23 Spans for a polygon. Extrema shown in black, interior pixels in gray. (a) Extrema computed by midpoint algorithm. (b) Extrema interior to polygon.

Samsung/Google, EX1027, p. 113

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

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

3.1 Given an intersection with an arbitrary, fractional x value, how do we determine which pixel on either side of that intersection is interior?

3.2 How do we deal with the special case of intersections at integer pixel coordinates?

3.3 How do we deal with the special case in 3.2 for shared vertices?

3.4 How do we deal with the special case in 3.2 in which the vertices define a horizontal edge?

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

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

Samsung/Google, EX1027, p. 114

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

3.6.1 Horizontal Edges

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

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

3.6.2 Slivers

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

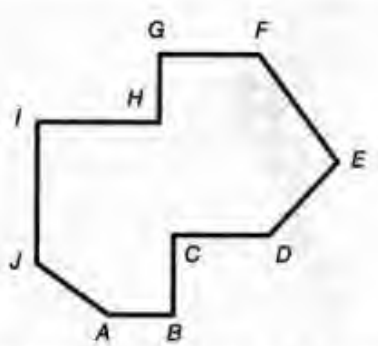


Fig. 3.24 Horizontal edges in a polygon.

Samsung/Google, EX1027, p. 115

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

3.6.3 Edge Coherence and the Scan-Line Algorithm

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

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

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

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

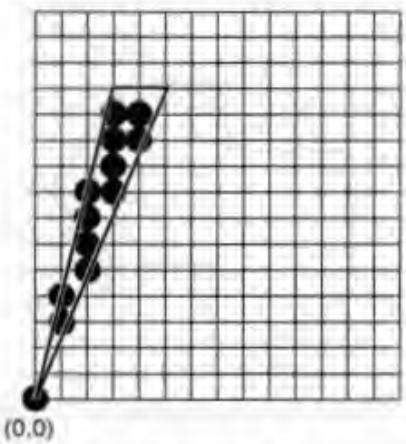


Fig. 3.25 Scan converting slivers of polygons.

Samsung/Google, EX1027, p. 116

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

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

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

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

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

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

Figure 3.26 Scan converting left edge of a polygon.

Samsung/Google, EX1027, p. 117

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

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

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

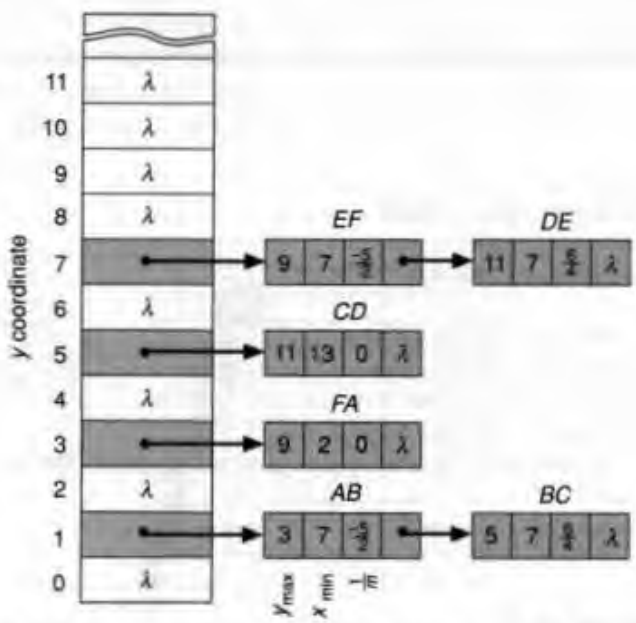


Fig. 3.27 Bucket-sorted edge table for polygon of Fig. 3.22.

Samsung/Google, EX1027, p. 118

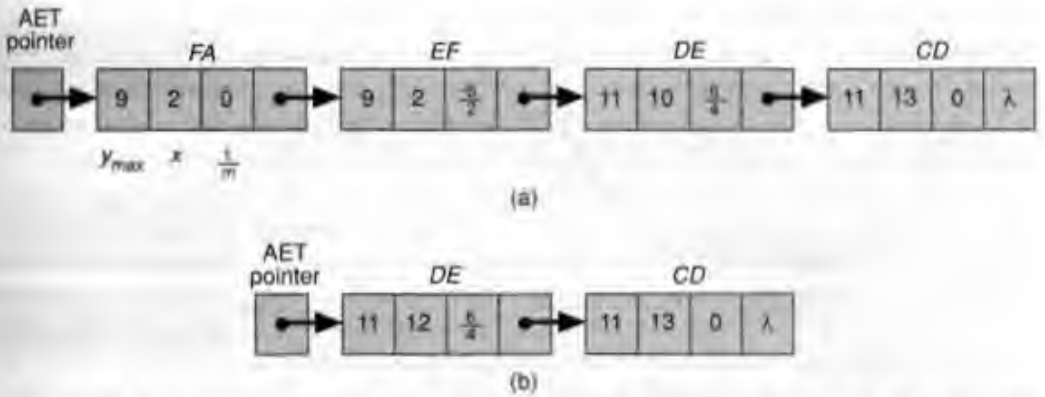


Fig. 3.28 Active-edge table for polygon of Fig. 3.22. (a) Scan line 9. (b) Scan line 10. (Note *DE*'s *x* coordinate in (b) has been rounded up for that left edge.)

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

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

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

3.7 FILLING ELLIPSE ARCS

The same general strategy of calculating spans for each scan line can be used for circles and ellipses as well. We accumulate span extrema for each iteration of the algorithm, rounding each extremum to ensure that the pixel is inside the region. As with scan converting the

Samsung/Google, EX1027, p. 119

as shown next. To write bitmap patterns transparently, we do a WritePixel with foreground color at a pixel for a 1 in the pattern, and we inhibit the WritePixel for a 0, as with line style. If, on the other hand, the bitmap pattern is applied in opaque mode, the 1s and 0s select foreground and background color, respectively.

The main issue for filling with patterns is the relation of the area of the pattern to that of the primitive. In other words, we need to decide where the pattern is "anchored" so that we know which pixel in the pattern corresponds to the current pixel of the primitive.

The first technique is to anchor the pattern at a vertex of a polygon by placing the leftmost pixel in the pattern's first row there. This choice allows the pattern to move when the primitive is moved, a visual effect that would be expected for patterns with a strong geometric organization, such as the cross-hatches often used in drafting applications. But there is no distinguished point on a polygon that is obviously right for such a relative anchor, and no distinguished points at all on smoothly varying primitives such as circles and ellipses. Therefore, the programmer must specify the anchor point as a point on or within the primitive. In some systems, the anchor point may even be applied to a group of primitives.

The second technique, used in SRGP, is to consider the entire screen as being tiled with the pattern and to think of the primitive as consisting of an outline or filled area of transparent bits that let the pattern show through. The standard position for such an absolute anchor is the screen origin. The pixels of the primitive are then treated as 1s that are **and**ed with the pattern. A side effect of this technique is that the pattern does not "stick to" the primitive if the primitive is moved slightly. Instead, the primitive moves as though it were a cutout on a fixed, patterned background, and thus its appearance may change as it is moved; for regular patterns without a strong geometric orientation, users may not even be aware of this effect. In addition to being computationally efficient, absolute anchoring allows primitives to overlap and abut seamlessly.

To apply the pattern to the primitive, we index it with the current pixel's (x, y) coordinates. Since patterns are defined as small M by N bitmaps or pixmaps, we use modular arithmetic to make the pattern repeat. The *pattern*[0, 0] pixel is considered coincident with the screen origin,[7] and we can write, for example, a bitmap pattern in transparent mode with the statement

```
if (pattern[x % M][y % N])
    WritePixel (x, y, value);
```

If we are filling an entire span in **replace** write mode, we can copy a whole row of the pattern at once, assuming a low-level version of a copyPixel facility is available to write multiple pixels. Let's say, for example, that the pattern is an 8 by 8 matrix. It thus repeats for every span of 8 pixels. If the leftmost point of a span is byte-aligned—that is, if the x value of the first pixel mod 8 is 0—then the entire first row of the pattern can be written out with a copyPixel of a 1 by 8 array; this procedure is repeated as many times as is necessary to fill the span. If either end of the span is not byte-aligned, the pixels not in the span must be masked out. Implementors spend much time making special cases of raster algorithms particularly efficient; for example, they test up-front to eliminate inner loops, and they write

[7] In window systems, the pattern is often anchored at the origin of the window coordinate system.

Samsung/Google, EX1027, p. 120

hand-tuned assembly-language code for inner loops that takes advantage of special hardware features such as instruction caches or particularly efficient loop instructions. This type of optimization is discussed in Chapter 19.

3.8.1 Pattern Filling Without Repeated Scan Conversion

So far, we have discussed filling in the context of scan conversion. Another technique is to scan convert a primitive first into a rectangular work area, and then to write each pixel from that bitmap to the appropriate place in the canvas. This so-called *rectangle write* to the canvas is simply a nested **for** loop in which a 1 writes the current color and a 0 writes nothing (for transparency) or writes the background color (for opacity). This two-step process is twice as much work as filling during scan conversion, and therefore is not worthwhile for primitives that are encountered and scan-converted only once. It pays off, however, for primitives that would otherwise be scan-converted repeatedly. This is the case for characters in a given font and size, which can be scan-converted ahead of time from their outlines. For characters defined only as bitmap fonts, or for other objects, such as icons and application symbols, that are painted or scanned in as bitmap images, scan conversion is not used in any case, and the rectangle write of their bitmaps is the only applicable technique. The advantage of a pre–scan-converted bitmap lies in the fact that it is clearly faster to write each pixel in a rectangular region, without having to do any clipping or span arithmetic, than to scan convert the primitive each time from scratch while doing such clipping.[8]

But since we have to write a rectangular bitmap into the canvas, why not just copyPixel the bitmap directly, rather than writing 1 pixel at a time? For bilevel displays, writing current color 1, copyPixel works fine: For transparent mode, we use **or** write mode; for opaque mode, we use **replace** write mode. For multilevel displays, we cannot write the bitmap directly with a single bit per pixel, but must convert each bit to a full n-bit color value that is then written.

Some systems have a more powerful copyPixel that can make copying subject to one or more source-read or destination-write masks. We can make good use of such a facility for transparent mode (used for characters in SRGP) if we can specify the bitmap as a destination-write mask and the source as an array of constant (current) color. Then, pixels are written in the current color only where the bitmap write mask has 1s; the bitmap write mask acts as an arbitrary clip region. In a sense, the explicit nested **for** loop for implementing the rectangle write on n-bits-per-pixel systems simulates this more powerful "copyPixel with write mask" facility.

Now consider another variation. We wish to draw a filled letter, or some other shape, not with a solid interior but with a patterned one. For example, we would like to create a thick letter "P" with a 50 percent gray stipple pattern (graying out the character), or a house icon with a two-tone brick-and-mortar pattern. How can we write such an object in opaque mode without having to scan convert it each time? The problem is that "holes" interior to the region where there are 0s in the bitmap should be written in background color, whereas holes outside the region (such as the cavity in the "P") must still be written

[8]There are added complications in the case of antialiasing, as discussed in Chapter 19.

Samsung/Google, EX1027, p. 121

transparently so as not to affect the image underneath. In other words, we want 0s in the shape's interior to signify background color, and 0s in its exterior, including any cavities, to belong to a write mask used to protect pixels outside the shape. If we scan convert on the fly, the problem that the 0s mean different things in different regions of the bitmap does not arise, because we never look at pixels outside the shape's boundary.

We use a four-step process to avoid repeated scan conversion, as shown in the mountain scene of Fig. 3.30. Using the outline of our icon (b), the first step is to create a "solid" bitmap to be used as a write mask/clipping region, with pixels interior to the object set to 1s, and those exterior set to 0s; this is depicted in (c), where white represents background pixels (0s) and black represents 1s. This scan conversion is done only once. As the second step, any time a patterned copy of the object is needed, we write the solid bitmap

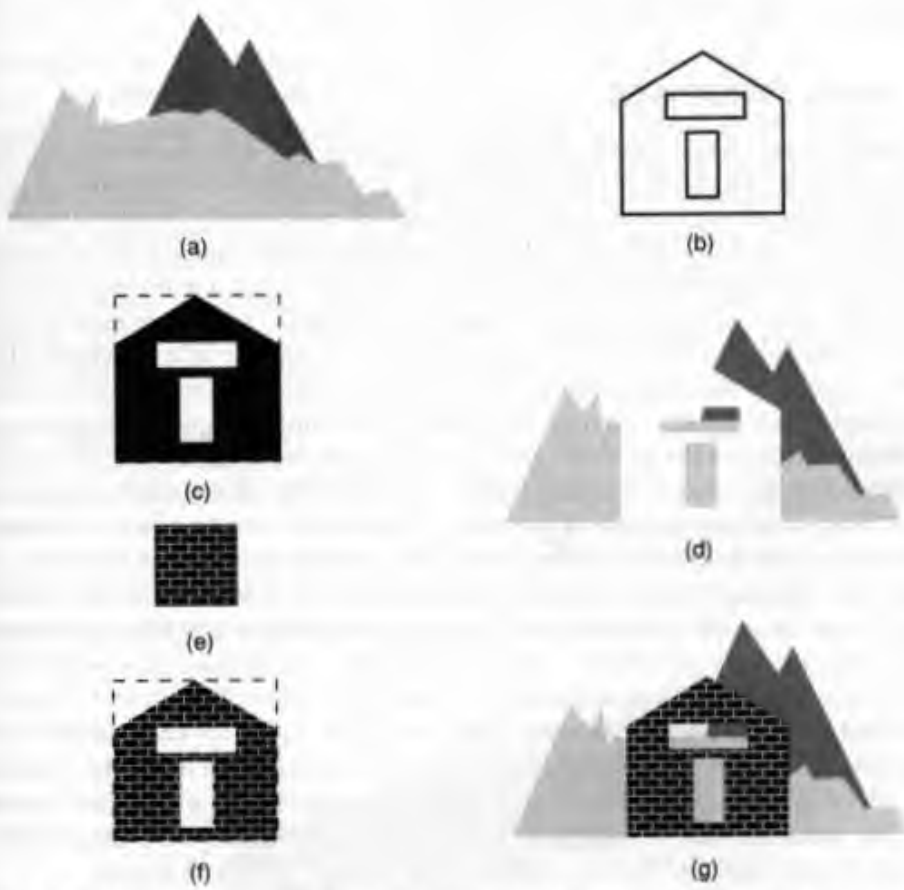


Fig. 3.30 Writing a patterned object in opaque mode with two transparent writes. (a) Mountain scene. (b) Outline of house icon. (c) Bitmap for solid version of house icon. (d) Clearing the scene by writing background. (e) Brick pattern. (f) Brick pattern applied to house icon. (g) Writing the screen transparently with patterned house icon.

Samsung/Google, EX1027, p. 122

transparently in background color to the canvas. This clears to background color a region of the shape of the object, as shown in (d), where the house-shaped region is set to white background within the existing mountain image. The third step is to create a patterned version of the object's solid bitmap by doing a copyPixel of a pattern rectangle (e) to the solid bitmap, using **and** mode. This turns some pixels internal to the object's shape from 1s to 0s (f), and can be seen as clipping out a piece of the arbitrarily large pattern in the shape of the object. Finally, we again write this new bitmap transparently to the same place in the canvas, but this time in the current, foreground color, as shown in (g). As in the first write to the canvas, all pixels outside the object's region are 0s, to protect pixels outside the region, whereas 0s inside the region do not affect the previously written (white) background; only where there are 1s is the (black) foreground written. To write the house with a solid red-brick pattern with gray mortar, we would write the solid bitmap in gray and the patterned bitmap in red; the pattern would have 1s everywhere except for small bands of 0s representing the mortar. In effect, we have reduced the rectangular write procedure that had to write two colors subject to a write mask to two write procedures that write transparently or copyPixel with a write mask.

3.9 THICK PRIMITIVES

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

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

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

Samsung/Google, EX1027, p. 123

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

3.9.1 Replicating Pixels

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

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

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

3.9.2 The Moving Pen

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

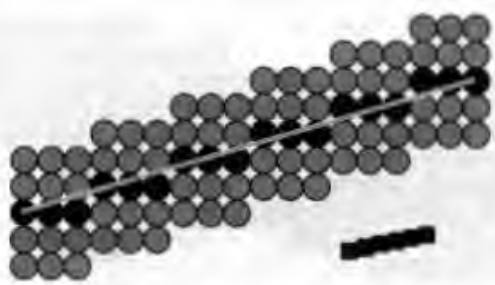


Fig. 3.31 Thick line drawn by column replication.

Samsung/Google, EX1027, p. 124

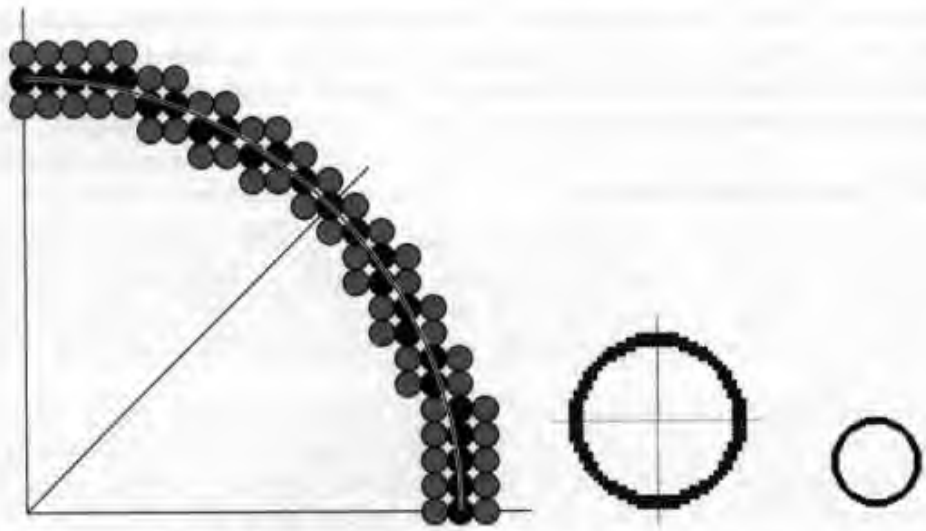


Fig. 3.32 Thick circle drawn by column replication.

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

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

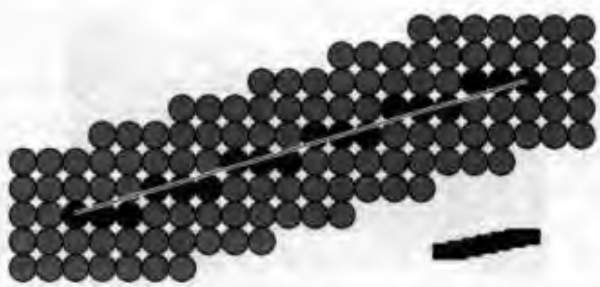


Fig. 3.33 Thick line drawn by tracing a rectangular pen.

Samsung/Google, EX1027, p. 125

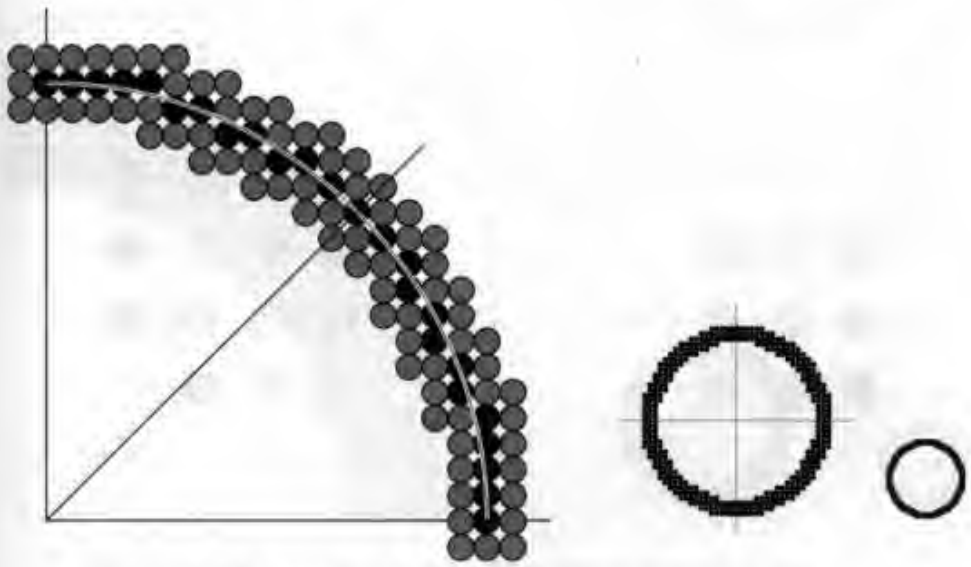


Fig. 3.34 Thick circle drawn by tracing a rectangular pen.

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

3.9.3 Filling Areas Between Boundaries

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

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

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

Samsung/Google, EX1027, p. 126

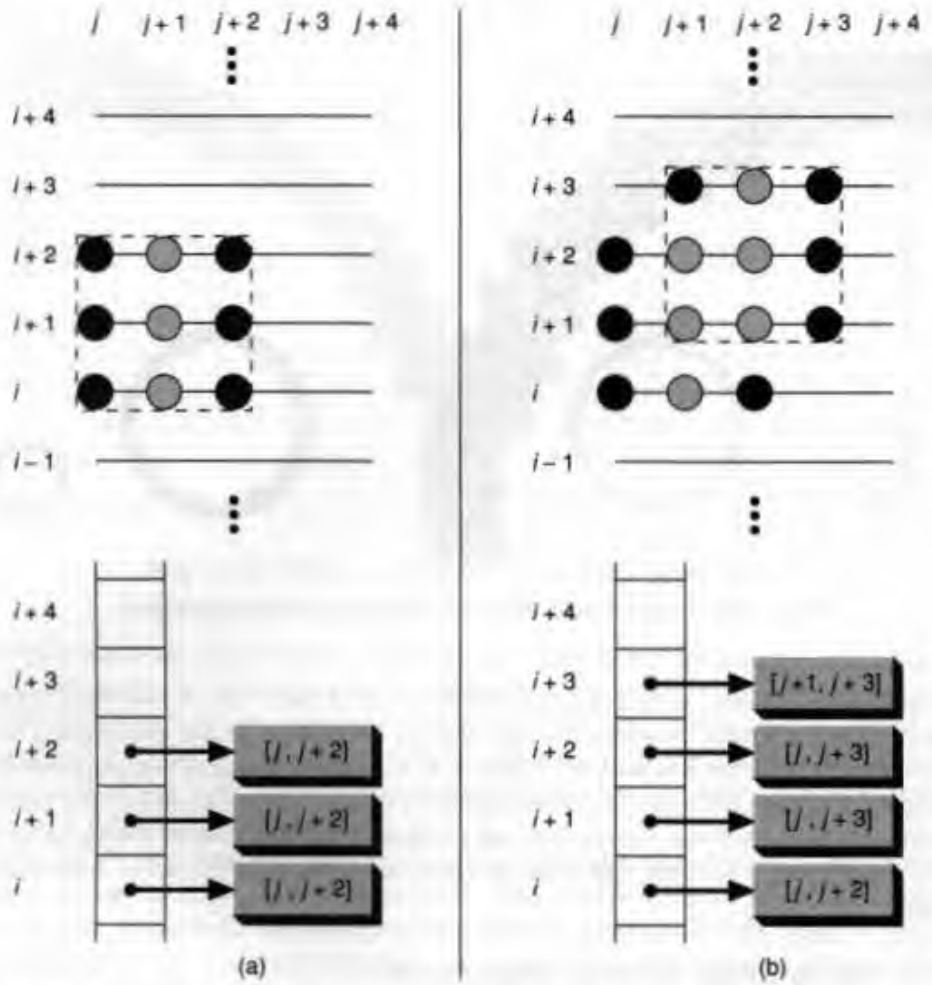


Fig. 3.35 Recording spans of the rectangular pen: (a) footprint at $x = j + 1$; (b) $x = j + 2$.

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

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

Samsung/Google, EX1027, p. 127

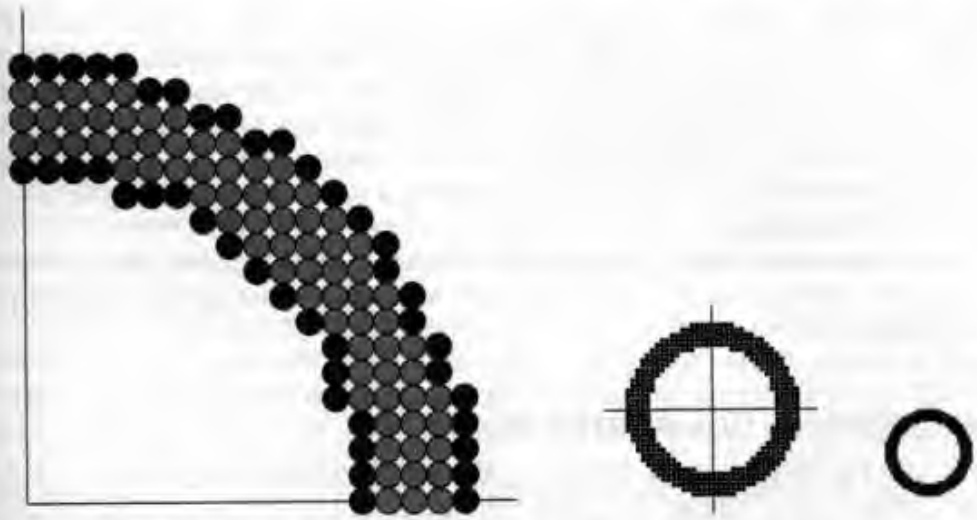


Fig. 3.36 Thick circle drawn by filling between concentric circles.

3.9.4 Approximation by Thick Polylines

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

3.10 LINE STYLE AND PEN STYLE

SRGP's line-style atribute can affect any outline primitive. In general, we must use conditional logic to test whether or not to write a pixel, writing only for 1s. We store the pattern write mask as a string of 16 **booleans** (e.g., a 16-bit **integer**); it should therefore repeat every 16 pixels. We modify the unconditional WritePixel statement in the line scan-conversion algorithm to handle this; for example,

```
if (bitstring[i % 16])
    WritePixel (x, y, value);
```

where the index i is a new variable incremented in the inner loop for this purpose. There is a drawback to this technique, however. Since each bit in the mask corresponds to an iteration of the loop, and not to a unit distance along the line, the length of dashes varies with the

Samsung/Google, EX1027, p. 128

angle of the line; a dash at an angle is longer than is a horizontal or vertical dash. For engineering drawings, this variation is unacceptable, and the dashes must be calculated and scan-converted as individual line segments of length invariant with angle. Thick lines are created as sequences of alternating solid and transparent rectangles whose vertices are calculated exactly as a function of the line style selected. The rectangles are then scan-converted individually; for horizontal and vertical lines, the program may be able to copyPixel the rectangle.

Line style and pen style interact in thick outline primitives. The line style is used to calculate the rectangle for each dash, and each rectangle is filled with the selected pen pattern (Fig. 3.37).

3.11 CLIPPING IN A RASTER WORLD

As we noted in the introduction to this chapter, it is essential that both clipping and scan conversion be done as rapidly as possible, in order to provide the user with quick updates resulting from changes to the application model. Clipping can be done analytically, on the fly during scan conversion, or as part of a copyPixel with the desired clip rectangle from a canvas storing unclipped primitives to the destination canvas. This third technique would be useful in situations where a large canvas can be generated ahead of time, and the user can then examine pieces of it for a significant period of time by panning the clip rectangle, without updating the contents of the canvas.

Combining clipping and scan conversion, sometimes called *scissoring*, is easy to do for filled or thick primitives as part of span arithmetic: Only the extrema need to be clipped, and no interior pixels need be examined. Scissoring shows yet another advantage of span coherence. Also, if an outline primitive is not much larger than the clip rectangle, not many pixels, relatively speaking, will fall outside the clip region. For such a case, it may well be faster to generate each pixel and to clip it (i.e., to write it conditionally) than to do analytical clipping beforehand. In particular, although the bounds test is in the inner loop, the expensive memory write is avoided for exterior pixels, and both the incremental computation and the testing may run entirely in a fast memory, such as a CPU instruction cache or a display controller's microcode memory.

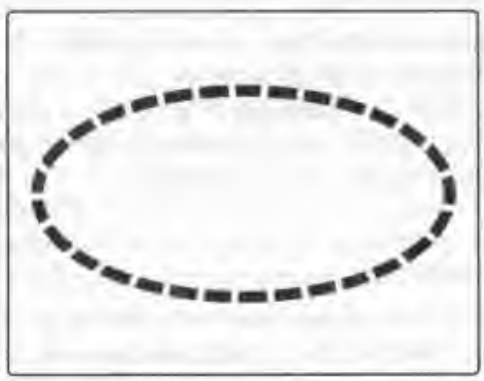


Fig. 3.37 Combining pen pattern and line style.

Samsung/Google, EX1027, p. 129

Other tricks may be useful. For example, one may "home in" on the intersection of a line with a clip edge by doing the standard midpoint scan-conversion algorithm on every ith pixel and testing the chosen pixel against the rectangle bounds until the first pixel that lies inside the region is encountered. Then the algorithm has to back up, find the first pixel inside, and to do the normal scan conversion thereafter. The last interior pixel could be similarly determined, or each pixel could be tested as part of the scan-conversion loop and scan conversion stopped the first time the test failed. Testing every eighth pixel works well, since it is a good compromise between having too many tests and too many pixels to back up (see Exercise 3.26).

For graphics packages that operate in floating point, it is best to clip analytically in the floating-point coordinate system and then to scan convert the clipped primitives, being careful to initialize decision variables correctly, as we did for lines in Section 3.2.3. For integer graphics packages such as SRGP, there is a choice between preclipping and then scan converting or doing clipping during scan conversion. Since it is relatively easy to do analytical clipping for lines and polygons, clipping of those primitives is often done before scan conversion, while it is faster to clip other primitives during scan conversion. Also, it is quite common for a floating-point graphics package to do analytical clipping in its coordinate system and then to call lower-level scan-conversion software that actually generates the clipped primitives; this integer graphics software could then do an additional raster clip to rectangular (or even arbitrary) window boundaries. Because analytic clipping of primitives is both useful for integer graphics packages and essential for 2D and 3D floating-point graphics packages, we discuss the basic analytical clipping algorithms in this chapter.

3.12 CLIPPING LINES

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

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

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

Samsung/Google, EX1027, p. 130

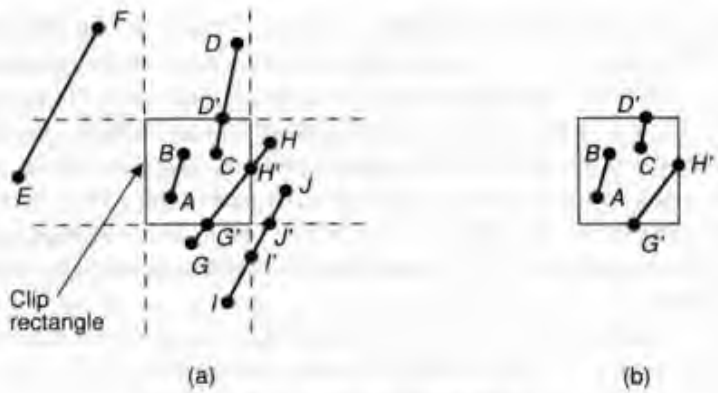


Fig. 3.38 Cases for clipping lines.

3.12.1 Clipping Endpoints

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

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

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

3.12.2 Clipping Lines by Solving Simultaneous Equations

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

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

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

Samsung/Google, EX1027, p. 131

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

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

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

3.12.3 The Cohen–Sutherland Line-Clipping Algorithm

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

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

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

Samsung/Google, EX1027, p. 132

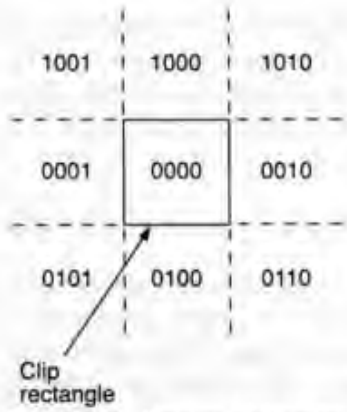


Fig. 3.39 Region outcodes.

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

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

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

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

Samsung/Google, EX1027, p. 133

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

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

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

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

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

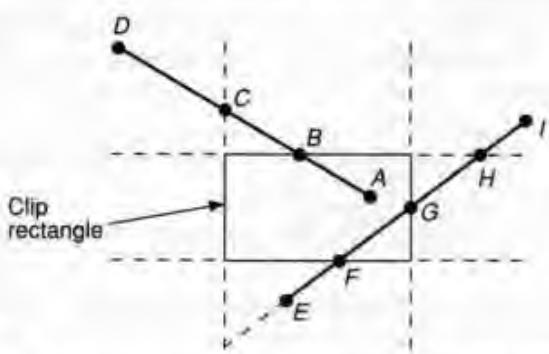


Fig. 3.40 Illustration of Cohen–Sutherland line clipping.

Samsung/Google, EX1027, p. 134

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

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

Fig. 3.41 *(Cont.).*

Samsung/Google, EX1027, p. 135

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

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

Fig. 3.41 Cohen–Sutherland line-clipping algorithm.

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

3.12.4 A Parametric Line-Clipping Algorithm

The Cohen–Sutherland algorithm is probably still the most commonly used line-clipping algorithm because it has been around longest and has been published widely. In 1978, Cyrus and Beck published an algorithm that takes a fundamentally different and generally more efficient approach to line clipping [CYRU78]. The Cyrus–Beck technique can be used to clip a 2D line against a rectangle or an arbitrary convex polygon in the plane, or a 3D line against an arbitrary convex polyhedron in 3D space. Liang and Barsky later independently developed a more efficient parametric line-clipping algorithm that is especially fast in the special cases of upright 2D and 3D clip regions [LIAN84]. In addition to taking advantage of these simple clip boundaries, they introduced more efficient trivial rejection tests that work for general clip regions. Here we follow the original Cyrus–Beck development to introduce parametric clipping. Since we are concerned only with upright clip rectangles, however, we reduce the Cyrus–Beck formulation to the more efficient Liang–Barsky case at the end of the development.

Recall that the Cohen–Sutherland algorithm, for lines that cannot be trivially accepted or rejected, calculates the (x, y) intersection of a line segment with a clip edge by substituting the known value of x or y for the vertical or horizontal clip edge, respectively. The parametric line algorithm, however, finds the value of the parameter t in the parametric

Samsung/Google, EX1027, p. 136

representation of the line segment for the point at which that segment intersects the infinite line on which the clip edge lies. Because all clip edges are in general intersected by the line, four values of t are calculated. A series of simple comparisons is used to determine which (if any) of the four values of t correspond to actual intersections. Only then are the (x, y) values for one or two actual intersections calculated. In general, this approach saves time over the Cohen–Sutherland intersection-calculation algorithm because it avoids the repetitive looping needed to clip to multiple clip-rectangle edges. Also, calculations in 1D parameter space are simpler than those in 3D coordinate space. Liang and Barsky improve on Cyrus–Beck by examining each t-value as it is generated, to reject some line segments before all four t-values have been computed.

The Cyrus–Beck algorithm is based on the following formulation of the intersection between two lines. Figure 3.42 shows a single edge E_i of the clip rectangle and that edge's outward normal N_i (i.e., outward to the clip rectangle[11]), as well as the line segment from P_0 to P_1 that must be clipped to the edge. Either the edge or the line segment may have to be extended to find the intersection point.

As before, this line is represented parametrically as

$$P(t) = P_0 + (P_1 - P_0)t,$$

where $t = 0$ at P_0 and $t = 1$ at P_1. Now, pick an arbitrary point P_{E_i} on edge E_i and consider the three vectors $P(t) - P_{E_i}$ from P_{E_i} to three designated points on the line from P_0 to P_1: the intersection point to be determined, an endpoint of the line on the inside halfplane of the edge, and an endpoint on the line in the outside halfplane of the edge. We can distinguish in which region a point lies by looking at the value of the dot product $N_i \cdot [P(t) - P_{E_i}]$. This value is negative for a point in the inside halfplane, zero for a point on the line containing the edge, and positive for a point that lies in the outside halfplane. The definitions of inside and outside halfplanes of an edge correspond to a counterclockwise enumeration of the edges of the clip region, a convention we shall use throughout this book. Now we can solve for the value of t at the intersection of P_0P_1 with the edge:

$$N_i \cdot [P(t) - P_{E_i}] = 0.$$

First, substitute for $P(t)$:

$$N_i \cdot [P_0 + (P_1 - P_0)t - P_{E_i}] = 0.$$

Next, group terms and distribute the dot product:

$$N_i \cdot [P_0 - P_{E_i}] + N_i \cdot [P_1 - P_0]t = 0.$$

Let $D = (P_1 - P_0)$ be the vector from P_0 to P_1, and solve for t:

$$t = \frac{N_i \cdot [P_0 - P_{E_i}]}{-N_i \cdot D}. \tag{3.1}$$

Note that this gives a valid value of t only if the denominator of the expression is nonzero.

[11]Cyrus and Beck use inward normals, but we prefer to use outward normals for consistency with plane normals in 3D, which are outward. Our formulation therefore differs only in the testing of a sign.

Samsung/Google, EX1027, p. 137

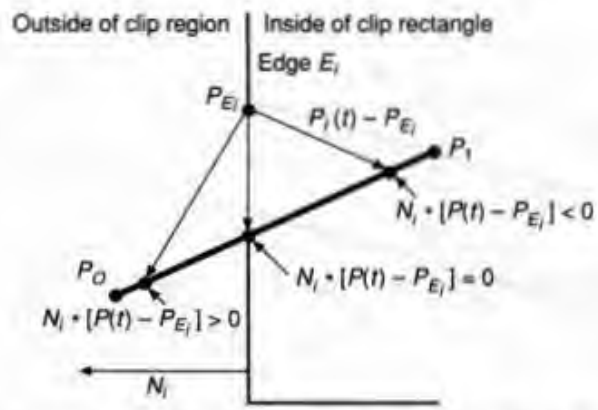


Fig. 3.42 Dot products for three points outside, inside, and on the boundary of the clip region.

For this to be true, the algorithm checks that

$N_i \neq 0$ (that is, the normal should not be 0; this could occur only as a mistake),

$D \neq 0$ (that is, $P_1 \neq P_0$),

$N_i \cdot D \neq 0$ (that is, the edge E_i and the line from P_0 to P_1 are not parallel. If they were parallel, there can be no single intersection for this edge, so the algorithm moves on to the next case.).

Equation (3.1) can be used to find the intersections between P_0P_1 and each edge of the clip rectangle. We do this calculation by determining the normal and an arbitrary P_{E_i}—say, an endpoint of the edge—for each clip edge, then using these values for all line segments. Given the four values of t for a line segment, the next step is to determine which (if any) of the values correspond to internal intersections of the line segment with edges of the clip rectangle. As a first step, any value of t outside the interval [0, 1] can be discarded, since it lies outside P_0P_1. Next, we need to determine whether the intersection lies on the clip boundary.

We could try simply sorting the remaining values of t, choosing the intermediate values of t for intersection points, as suggested in Fig. 3.43 for the case of line 1. But how do we distinguish this case from that of line 2, in which no portion of the line segment lies in the clip rectangle and the intermediate values of t correspond to points not on the clip boundary? Also, which of the four intersections of line 3 are the ones on the clip boundary?

The intersections in Fig. 3.43 are characterized as "potentially entering" (PE) or "potentially leaving" (PL) the clip rectangle, as follows: If moving from P_0 to P_1 causes us to cross a particular edge to enter the edge's inside halfplane, the intersection is PE; if it causes us to leave the edge's inside halfplane, it is PL. Notice that, with this distinction, two interior intersection points of a line intersecting the clip rectangle have opposing labels.

Formally, intersections can be classified as PE or PL on the basis of the angle between P_0P_1 and N_i: If the angle is less than 90°, the intersection is PL; if it is greater than 90°, it is PE. This information is contained in the sign of the dot product of N_i and P_0P_1:

Samsung/Google, EX1027, p. 138

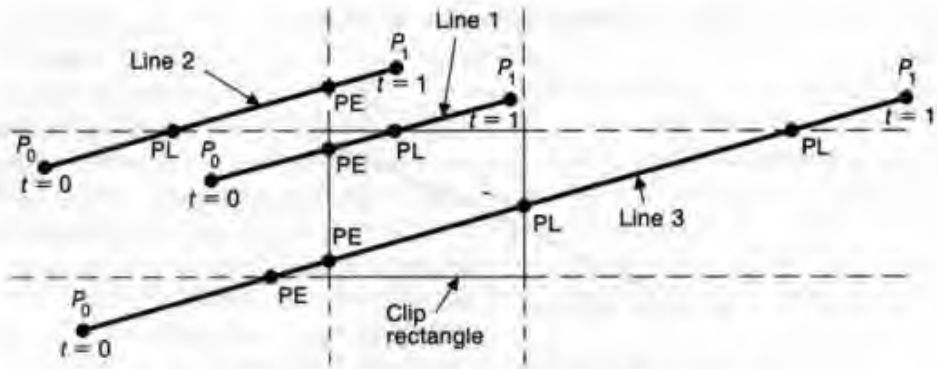


Fig. 3.43 Lines lying diagonal to the clip rectangle.

$$N_i \cdot D < 0 \Rightarrow \text{PE (angle greater than 90)},$$
$$N_i \cdot D > 0 \Rightarrow \text{PL (angle less than 90)}.$$

Notice that $N_i \cdot D$ is merely the denominator of Eq. (3.1), which means that, in the process of calculating t, the intersection can be trivially categorized.

With this categorization, line 3 in Fig. 3.43 suggests the final step in the process. We must choose a (PE, PL) pair that defines the clipped line. The portion of the infinite line through P_0P_1 that is within the clipping region is bounded by the PE intersection with the largest t value, which we call t_E, and the PL intersection with the smallest t value, t_L. The intersecting line segment is then defined by the range (t_E, t_L). But because we are interested in intersecting P_0P_1, not the infinite line, the definition of the range must be further modified so that $t = 0$ is a lower bound for t_E and $t = 1$ is an upper bound for t_L. What if $t_E > t_L$? This is exactly the case for line 2. It means that no portion of P_0P_1 is within the clip rectangle, and the entire line is rejected. Values of t_E and t_L that correspond to actual intersections are used to calculate the corresponding x and y coordinates.

The completed algorithm for upright clip rectangles is pseudocoded in Fig. 3.44. Table 3.1 shows for each edge the values of N_i, a canonical point on the edge, P_{E_i}, the vector $P_0 - P_{E_i}$ and the parameter t. Interestingly enough, because one coordinate of each normal is 0, we do not need to pin down the corresponding coordinate of P_{E_i} (denoted by an indeterminate x or y). Indeed, because the clip edges are horizontal and vertical, many simplifications apply that have natural interpretations. Thus we see from the table that the numerator, the dot product $N_i \cdot (P_0 - P_{E_i})$ determining whether the endpoint P_0 lies inside or outside a specified edge, reduces to the directed horizontal or vertical distance from the point to the edge. This is exactly the same quantity computed for the corresponding component of the Cohen-Sutherland outcode. The denominator dot product $N_i \cdot D$, which determines whether the intersection is potentially entering or leaving, reduces to $\pm dx$ or dy: if dx is positive, the line moves from left to right and is PE for the left edge, PL for the right edge, and so on. Finally, the parameter t, the ratio of numerator and denominator, reduces to the distance to an edge divided by dx or dy, exactly the constant of proportionality we could calculate directly from the parametric line formulation. Note that it is important to preserve the signs of the numerator and denominator instead of cancelling minus signs, because the numerator and denominator as signed distances carry information that is used in the algorithm.

Samsung/Google, EX1027, p. 139

```
precalculate N_i and select a P_Ei for each edge;

for (each line segment to be clipped) {
    if (P1 == P0)
        line is degenerate so clip as a point;
    else {
        t_E = 0; t_L = 1;
        for (each candidate intersection with a clip edge) {
            if (N_i • D != 0) {      /* Ignore edges parallel to line for now */
                calculate t;
                use sign of N_i • D to categorize as PE or PL;
                if (PE) t_E = max (t_E, t);
                if (PL) t_L = min (t_L, t);
            }
        }
        if (t_E > t_L)
            return NULL;
        else
            return P(t_E) and P(t_L) as true clip intersections;
    }
}
```

Fig. 3.44 Pseudocode for Cyrus–Beck parametric line clipping algorithm.

The complete version of the code, adapted from [LIAN84] is shown in Fig. 3.45. The procedure calls an internal function, CLIPt(), that uses the sign of the denominator to determine whether the line segment-edge intersection is potentially entering (PE) or leaving (PL), computes the parameter value of the intersection and checks to see if trivial rejection can be done because the new value of t_E or t_L would cross the old value of t_L or t_E, respectively. It also signals trivial rejection if the line is parallel to the edge and on the

TABLE 3.1 CALCULATIONS FOR PARAMETRIC LINE CLIPPING ALGORITHM*

Clip edge$_i$	Normal N_i	P_{E_i}	$P_0 - P_{E_i}$	$t = \frac{N_i \cdot (P_0 - P_{E_i})}{-N_i \cdot D}$
left: $x = x_{min}$	$(-1, 0)$	(x_{min}, y)	$(x_0 - x_{min}, y_0 - y)$	$\frac{-(x_0 - x_{min})}{(x_1 - x_0)}$
right: $x = x_{max}$	$(1, 0)$	(x_{max}, y)	$(x_0 - x_{max}, y_0 - y)$	$\frac{(x_0 - x_{max})}{-(x_1 - x_0)}$
bottom: $y = y_{min}$	$(0, -1)$	(x, y_{min})	$(x_0 - x, y_0 - y_{min})$	$\frac{-(y_0 - y_{min})}{(y_1 - y_0)}$
top: $y = y_{max}$	$(0, 1)$	(x, y_{max})	$(x_0 - x, y_0 - y_{max})$	$\frac{(y_0 - y_{max})}{-(y_1 - y_0)}$

*The exact coordinates of the point P_{E_i} on each edge are irrelevant to the computation, so they have been denoted by variables x and y. For a point on the right edge, $x = x_{min}$ as indicated in the first row, third entry.

Samsung/Google, EX1027, p. 140

```
void Clip2D (double *x0, double *y0, double *x1, double *y1, boolean *visible)
/* Clip 2D line segment with endpoints (x0, y0) and (x1, y1), against upright */
/* clip rectangle with corners at (xmin, ymin) and (xmax, ymax); these are */
/* globals or could be passed as parameters also. The flag visible is set TRUE. */
/* if a clipped segment is returned in endpoint parameters. If the line */
/* is rejected, the endpoints are not changed and visible is set to FALSE. */
{
    double dx = *x1 - *x0;
    double dy = *y1 - *y0;
    /* Output is generated only if line is inside all four edges. */
    *visible = FALSE;
    /* First test for degenerate line and clip the point; ClipPoint returns */
    /* TRUE if the point lies inside the clip rectangle. */
    if (dx == 0 && dy == 0 && ClipPoint (*x0, *y0))
        *visible = TRUE;
    else {
        double tE = 0.0;
        double tL = 1.0;
        if (CLIPt (dx, xmin - *x0, &tE, &tL))                    /* Inside wrt left edge */
            if (CLIPt (-dx, *x0 - xmax, &tE, &tL))               /* Inside wrt right edge */
                if (CLIPt (dy, ymin - *y0, &tE, &tL))            /* Inside wrt bottom edge */
                    if (CLIPt (-dy, *y0 - ymax, &tE, &tL)) {     /* Inside wrt top edge */
                        *visible = TRUE;
                        /* Compute PL intersection, if tL has moved */
                        if (tL < 1) {
                            *x1 = *x0 + tL * dx;
                            *y1 = *y0 + tL * dy;
                        }
                        /* Compute PE intersection, if tE has moved */
                        if (tE > 0) {
                            *x0 += tE * dx;
                            *y0 += tE * dy;
                        }
                    }
    }
} /* Clip2D */

boolean CLIPt (double denom, double num, double *tE, double *tL)
/* This function computes a new value of tE or tL for an interior intersection */
/* of a line segment and an edge. Parameter denom is -(N_i • D), which reduces to */
/* ± Δx, Δy for upright rectangles (as shown in Table 3.1); its sign */
/* determines whether the intersection is PE or PL. Parameter num is N_i • (P_0 - P_Ei) */
/* for a particular edge/line combination, which reduces to directed horizontal */
/* and vertical distances from P_0 to an edge; its sign determines visibility */
/* of P_0 and is used to trivially reject horizontal or vertical lines. If the */
/* line segment can be trivially rejected, FALSE is returned; if it cannot be, */
/* TRUE is returned and the value of tE or tL is adjusted, if needed, for the */
/* portion of the segment that is inside the edge. */
```

Fig. 3.45 *(Cont.)*.

Samsung/Google, EX1027, p. 141

```
{
    double t;

    if (denom > 0) {            /* PE intersection */
        t = num / denom;        /* Value of t at the intersection */
        if (t > tL)             /* tE and tL crossover */
            return FALSE;       /* so prepare to reject line */
        else if (t > tE)        /* A new tE has been found */
            tE = t;
    } else if (denom < 0) {     /* PL intersection */
        t = num / denom;        /* Value of t at the intersection */
        if (t < tE)             /* tE and tL crossover */
            return FALSE;       /* so prepare to reject line */
        else                    /* A new tL has been found */
            tL = t;
    } else if (num > 0)         /* Line on outside of edge */
        return FALSE;
    return TRUE;
}  /* CLIPt */
```

Fig. 3.45 Code for Liang–Barsky parametric line-clipping algorithm.

outside; i.e., would be invisible. The main procedure then does the actual clipping by moving the endpoints to the most recent values of t_E and t_L computed, but only if there is a line segment inside all four edges. This condition is tested for by a four-deep nested **if** that checks the flags returned by the function signifying whether or not the line segment was rejected.

In summary, the Cohen–Sutherland algorithm is efficient when outcode testing can be done cheaply (for example, by doing bitwise operations in assembly language) and trivial acceptance or rejection is applicable to the majority of line segments. Parametric line clipping wins when many line segments need to be clipped, since the actual calculation of the coordinates of the intersection points is postponed until needed, and testing can be done on parameter values. This parameter calculation is done even for endpoints that would have been trivially accepted in the Cohen–Sutherland strategy, however. The Liang–Barsky algorithm is more efficient than the Cyrus–Beck version because of additional trivial rejection testing that can avoid calculation of all four parameter values for lines that do not intersect the clip rectangle. For lines that cannot be trivially rejected by Cohen–Sutherland because they do not lie in an invisible halfplane, the rejection tests of Liang–Barsky are clearly preferable to the repeated clipping required by Cohen–Sutherland. The Nicholl et

Samsung/Google, EX1027, p. 142

al. algorithm of Section 19.1.1 is generally preferable to either Cohen–Sutherland or Liang–Barsky but does not generalize to 3D, as does parametric clipping. Speed-ups to Cohen–Sutherland are discussed in [DUVA90]. Exercise 3.29 concerns instruction counting for the two algorithms covered here, as a means of contrasting their efficiency under various conditions.

3.13 CLIPPING CIRCLES AND ELLIPSES

To clip a circle against a rectangle, we can first do a trivial accept/reject test by intersecting the circle's extent (a square of the size of the circle's diameter) with the clip rectangle, using the algorithm in the next section for polygon clipping. If the circle intersects the rectangle, we divide it into quadrants and do the trivial accept/reject test for each. These tests may lead in turn to tests for octants. We can then compute the intersection of the circle and the edge analytically by solving their equations simultaneously, and then scan convert the resulting arcs using the appropriately initialized algorithm with the calculated (and suitably rounded) starting and ending points. If scan conversion is fast, or if the circle is not too large, it is probably more efficient to scissor on a pixel-by-pixel basis, testing each boundary pixel against the rectangle bounds before it is written. An extent check would certainly be useful in any case. If the circle is filled, spans of adjacent interior pixels on each scan line can be filled without bounds checking by clipping each span and then filling its interior pixels, as discussed in Section 3.7.

To clip ellipses, we use extent testing at least down to the quadrant level, as with circles. We can then either compute the intersections of ellipse and rectangle analytically and use those (suitably rounded) endpoints in the appropriately initialized scan-conversion algorithm given in the next section, or clip as we scan convert.

3.14 CLIPPING POLYGONS

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

3.14.1 The Sutherland–Hodgman Polygon-Clipping Algorithm

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

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

Samsung/Google, EX1027, p. 143

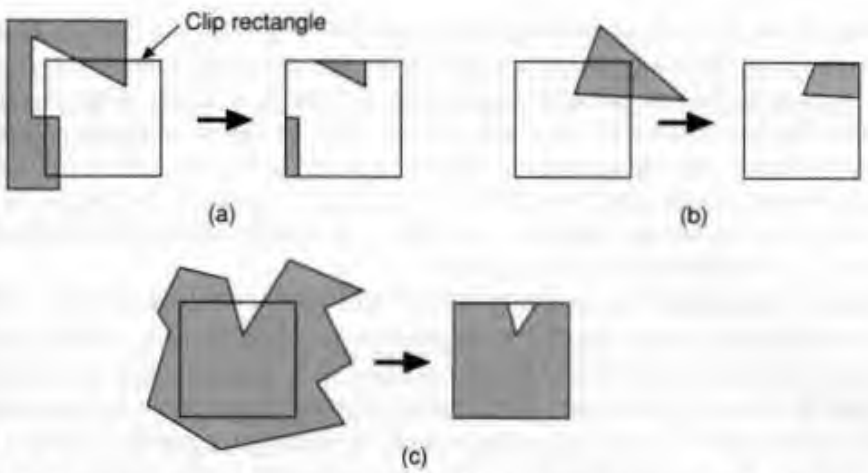


Fig. 3.46 Examples of polygon clipping. (a) Multiple components. (b) Simple convex case. (c) Concave case with many exterior edges.

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

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

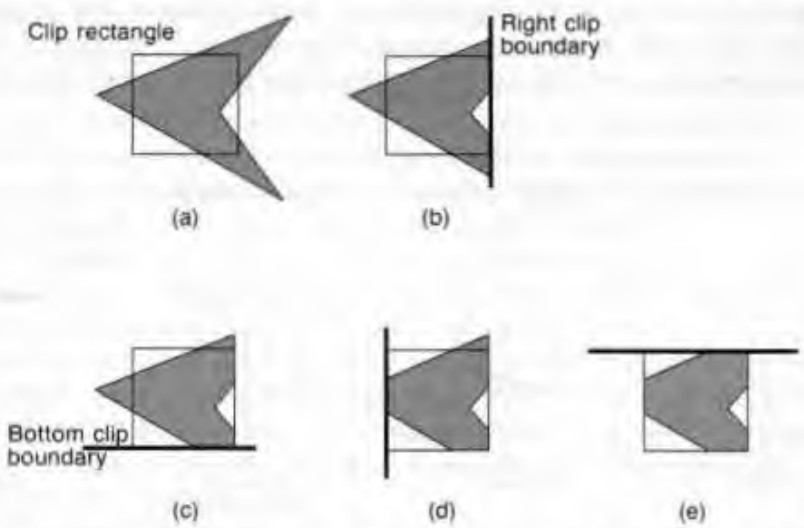


Fig. 3.47 Polygon clipping, edge by edge. (a) Before clipping. (b) Clip on right. (c) Clip on bottom. (d) Clip on left. (e) Clip on top; polygon is fully clipped.

Samsung/Google, EX1027, p. 144

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

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

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

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

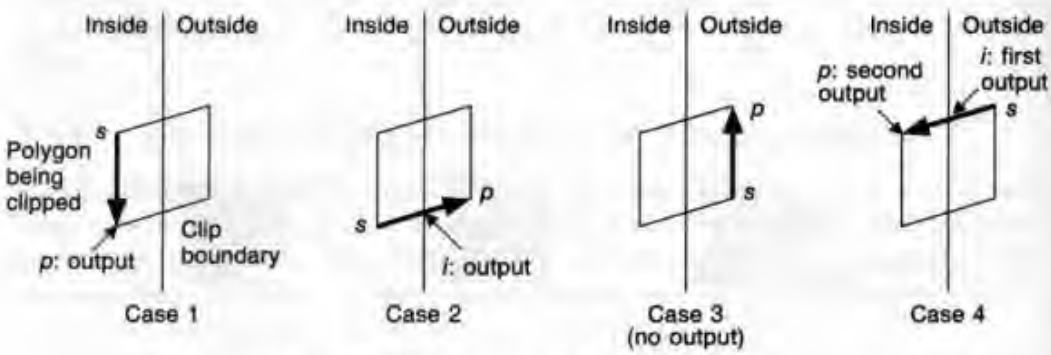


Fig. 3.48 Four cases of polygon clipping.

Samsung/Google, EX1027, p. 145

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

3.15 GENERATING CHARACTERS

3.15.1 Defining and Clipping Characters

There are two basic techniques for defining characters. The most general but most computationally expensive way is to define each character as a curved or polygonal outline and to scan convert it as needed. We first discuss the other, simpler way, in which each character in a given font is specified as a small rectangular bitmap. Generating a character then entails simply using a copyPixel to copy the character's image from an offscreen canvas, called a *font cache*, into the frame buffer at the desired position.

The font cache may actually be in the frame buffer, as follows. In most graphics systems in which the display is refreshed from a private frame buffer, that memory is larger than is strictly required for storing the displayed image. For example, the pixels for a rectangular screen may be stored in a square memory, leaving a rectangular strip of "invisible" screen memory. Alternatively, there may be enough memory for two screens, one of which is being refreshed and one of which is being drawn in, to double-buffer the image. The font cache for the currently displayed font(s) is frequently stored in such invisible screen memory because the display controller's copyPixel works fastest within local image memory. A related use for such invisible memory is for saving screen areas temporarily obscured by popped-up images, such as windows, menus, and forms.

The bitmaps for the font cache are usually created by scanning in enlarged pictures of characters from typesetting fonts in various sizes; a typeface designer can then use a paint program to touch up individual pixels in each character's bitmap as necessary. Alternatively, the type designer may use a paint program to create, from scratch, fonts that are especially designed for screens and low-resolution printers. Since small bitmaps do not scale well, more than one bitmap must be defined for a given character in a given font just to provide various standard sizes. Furthermore, each type face requires its own set of bitmaps. Therefore, a distinct font cache is needed for each font loaded by the application.

Bitmap characters are clipped automatically by SRGP as part of its implementation of copyPixel. Each character is clipped to the destination rectangle on a pixel-by-pixel basis, a technique that lets us clip a character at any row or column of its bitmap. For systems with slow copyPixel operations, a much faster, although cruder, method is to clip the character or even the entire string on an all-or-nothing basis by doing a trivial accept of the character or string extent. Only if the extent is trivially accepted is the copyPixel applied to the character or string. For systems with a fast copyPixel, it is still useful to do trivial accept/reject testing of the string extent as a precursor to clipping individual characters during the copyPixel operation.

SRGP's simple bitmap font-cache technique stores the characters side by side in a canvas that is quite wide but is only as tall as the tallest character; Fig. 3.50 shows a portion

Samsung/Google, EX1027, p. 146

```
typedef point vertex;                          /* point holds double x, y */
typedef vertex edge[2];
typedef vertex vertexArray[MAX];               /* MAX is a declared constant */

static void Output (vertex, int *, vertexArray);
static boolean Inside (vertex, edge);
static vertex Intersect (vertex, vertex, edge);

void SutherlandHodgmanPolygonClip (
        vertexArray inVertexArray,             /* Input vertex array */
        vertexArray outVertexArray,            /* Output vertex array */
        int inLength,                          /* No. of entries in inVertexArray */
        int *outLength,                        /* No. of entries in outVertexArray */
        edge clipBoundary)                     /* Edge of clip polygon */
{
    vertex s, p,                               /* Start, end pt. of current polygon edge */
    i;                                         /* Intersection pt. with a clip boundary */
    int j;                                     /* Vertex loop counter */

    *outLength = 0;   /* Start with the last vertex in inVertexArray */
    s = inVertexArray[inLength - 1];
    for (j = 0; j < inLength; j++) {
        p = inVertexArray[j];   /* Now s and p correspond to the vertices in Fig. 3.48 */
        if (Inside (p, clipBoundary)) {            /* Cases 1 and 4 */
            if (Inside (s, clipBoundary))          /* Case 1 */
                Output (p, outLength, outVertexArray);
            else {                                 /* Case 4 */
                i = Intersect (s, p, clipBoundary);
                Output (i, outLength, outVertexArray);
                Output (p, outLength, outVertexArray);
            }
        } else                                     /* Cases 2 and 3 */
            if (Inside (s, clipBoundary)) {        /* Case 2 */
                i = Intersect (s, p, clipBoundary);
                Output (i, outLength, outVertexArray);
            }                                      /* No action for case 3 */
        s = p;                                     /* Advance to next pair of vertices */
    } /* for */
} /* SutherlandHodgmanPolygonClip */

/* Adds newVertex to outVertexArray and then updates outLength */
static void Output (vertex newVertex, int *outLength, vertexArray outVertexArray)
{
...
}
/* Checks whether the vertex lies inside the clip edge or not */
static boolean Inside (vertex testVertex, edge clipBoundary)
```

Fig. 3.49 *(Cont.)*.

Samsung/Google, EX1027, p. 147

```
{
...
}

/* Clips polygon edge (first, second) against clipBoundary, outputs the new point */
static vertex Intersect (vertex first, vertex second, edge clipBoundary)
{
...
}
```

Fig. 3.49 Sutherland–Hodgman polygon-clipping algorithm.

of the cache, along with discrete instances of the same characters at low resolution. Each loaded font is described by a struct (declared in Fig. 3.51) containing a reference to the canvas that stores the characters' images, along with information on the height of the characters and the amount of space to be placed between adjacent characters in a text string. (Some packages store the space between characters as part of a character's width, to allow variable intercharacter spacing.)

As described in Section 2.1.5, descender height and total height are constants for a given font—the former is the number of rows of pixels at the bottom of the font cache used only by descenders, and the latter is simply the height of the font-cache canvas. The width of a character, on the other hand, is not considered a constant; thus, a character can occupy the space that suits it, rather than being forced into a fixed-width character box. SRGP puts a fixed amount of space between characters when it draws a text string, the amount being specified as part of each font's descriptor. A word-processing application can display lines of text by using SRGP to display individual words of text, and can right-justify lines by using variable spacing between words and after punctuation to fill out lines so that their rightmost characters are aligned at the right margin. This involves using the text-extent inquiry facilities to determine where the right edge of each word is, in order to calculate the start of the next word. Needless to say, SRGP's text-handling facilities are really too crude for sophisticated word processing, let alone for typesetting programs, since such applications require far finer control over the spacing of individual letters to deal with such effects as sub- and superscripting, kerning, and printing text that is not horizontally aligned.

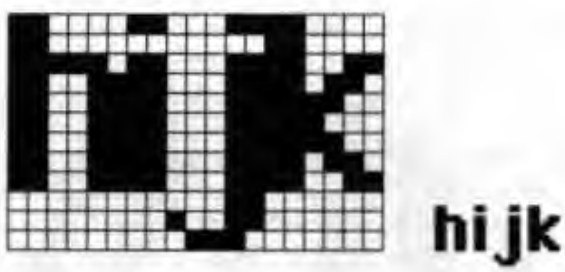


Fig. 3.50 Portion of an example of a font cache.

Samsung/Google, EX1027, p. 148

```
typedef struct {
    int leftX, width;                    /* Horizontal location, width of image in font cache */
} charLocation;

typedef struct {
    canvasID cache;
    int descenderHeight, totalHeight;    /* Height is a constant; width varies */
    int interCharacterSpacing;           /* Measured in pixels */
    charLocation locationTable[128];     /* Explained in the text */
} fontCacheDescriptor;
```

Fig. 3.51 Type declarations for the font cache.

3.15.2 Implementation of a Text Output Primitive

In the code of Fig. 3.52, we show how SRGP text is implemented internally: Each character in the given string is placed individually, and the space between characters is dictated by the appropriate field in the font descriptor. Note that complexities such as dealing with mixed fonts in a string must be handled by the application program.

```
void SRGP_characterText (
        point origin,                    /* Where to place the character in the current canvas */
        char *stringToPrint,
        fontCacheDescriptor fontInfo)
{
    int i;

    /* Origin specified by the application is for baseline and does not include descender. */
    origin.y -= fontInfo.descenderHeight;

    for (i = 0; i < strlen (stringToPrint); i++) {
        rectangle fontCacheRectangle;
        char charToPrint = stringToPrint[i];
        /* Find the rectangular region within the cache wherein the character lies */
        charLocation *fip = &fontInfo.locationTable[charToPrint];

        fontCacheRectangle.bottomLeft = SRGP_defPoint (fip->leftX, 0);
        fontCacheRectangle.topRight = SRGP_defPoint (fip->leftX + fip->width - 1,
                                                     fontInfo.totalHeight - 1);

        SRGP_copyPixel (fontInfo.cache, fontCacheRectangle, origin);
        /* Update the origin to move past the new character plus intercharacter spacing */
        origin.x += fip->width + interCharacterSpacing;
    }
}  /* SRGP_characterText */
```

Fig. 3.52 Implementation of character placement for SRGP's text primitive.

Samsung/Google, EX1027, p. 149

We mentioned that the bitmap technique requires a distinct font cache for each combination of font, size, and face for each different resolution of display or output device supported. A single font in eight different point sizes and four faces (normal, bold, italic, bold italic) thus requires 32 font caches! Figure 3.53(a) shows a common way of allowing a single font cache to support multiple face variations: The italic face is approximated by splitting the font's image into regions with horizontal "cuts," then offsetting the regions while performing a sequence of calls to SRGP_copyPixel.

This crude approximation does not make pleasing characters; for example, the dot over the "*i*" is noncircular. More of a problem for the online user, the method distorts the intercharacter spacing and makes picking much more difficult. A similar trick to achieve boldface is to copy the image twice in **or** mode with a slight horizontal offset (Fig. 3.53b). These techniques are not particularly satisfactory, in that they can produce illegible characters, especially when combined with subscripting.

A better way to solve the storage problem is to store characters in an abstract, device-independent form using polygonal or curved outlines of their shapes defined with floating-point parameters, and then to transform them appropriately. Polynomial functions called *splines* (see Chapter 11) provide smooth curves with continuous first and higher derivatives and are commonly used to encode text outlines. Although each character definition takes up more space than its representation in a font cache, multiple sizes may be derived from a single stored representation by suitable scaling; also, italics may be quickly approximated by shearing the outline. Another major advantage of storing characters in a completely device-independent form is that the outlines may be arbitrarily translated, rotated, scaled, and clipped (or used as clipping regions themselves).

The storage economy of splined characters is not quite so great as this description suggests. For instance, not all point sizes for a character may be obtained by scaling a single abstract shape, because the shape for an aesthetically pleasing font is typically a function of point size; therefore, each shape suffices for only a limited range of point sizes. Moreover, scan conversion of splined text requires far more processing than the simple copyPixel implementation, because the device-independent form must be converted to pixel coordinates on the basis of the current size, face, and transformation attributes. Thus, the font-cache technique is still the most common for personal computers and even is used for many workstations. A strategy that offers the best of both methods is to store the fonts in outline form but to convert the ones being used in a given application to their bitmap equivalents—for example, to build a font cache on the fly. We discuss processing of splined text in more detail in Section 19.4.

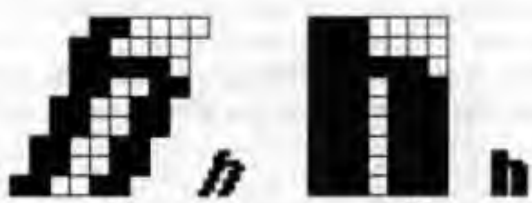


Fig. 3.53 Tricks for creating different faces for a font. (a) Italic. (b) Bold.

Samsung/Google, EX1027, p. 150

3.16 SRGP_copyPixel

If only WritePixel and ReadPixel low-level procedures are available, the SRGP_copyPixel procedure can be implemented as a doubly nested **for** loop for each pixel. For simplicity, assume first that we are working with a bilevel display and do not need to deal with the low-level considerations of writing bits that are not word-aligned; in Section 19.6, we cover some of these more realistic issues that take hardware-memory organization into account. In the inner loop of our simple SRGP_copyPixel, we do a ReadPixel of the source and destination pixels, logically combine them according to the SRGP write mode, and then WritePixel the result. Treating **replace** mode, the most common write mode, as a special case allows a simpler inner loop that does only a ReadPixel/WritePixel of the source into the destination, without having to do a logical operation. The clip rectangle is used during address calculation to restrict the region into which destination pixels are written.

3.17 ANTIALIASING

3.17.1 Increasing Resolution

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

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

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

3.17.2 Unweighted Area Sampling

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

Samsung/Google, EX1027, p. 151

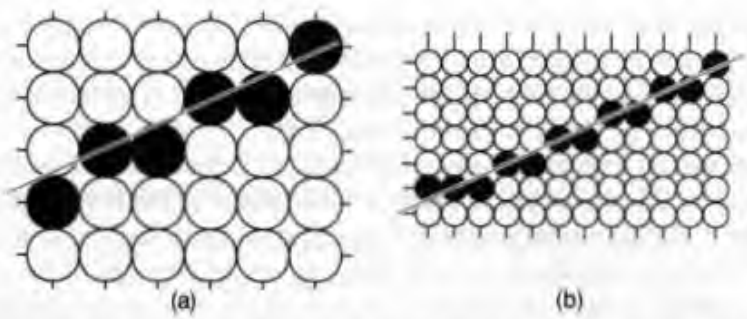


Fig. 3.54 (a) Standard midpoint line on a bilevel display. (b) Same line on a display that has twice the linear resolution.

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

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

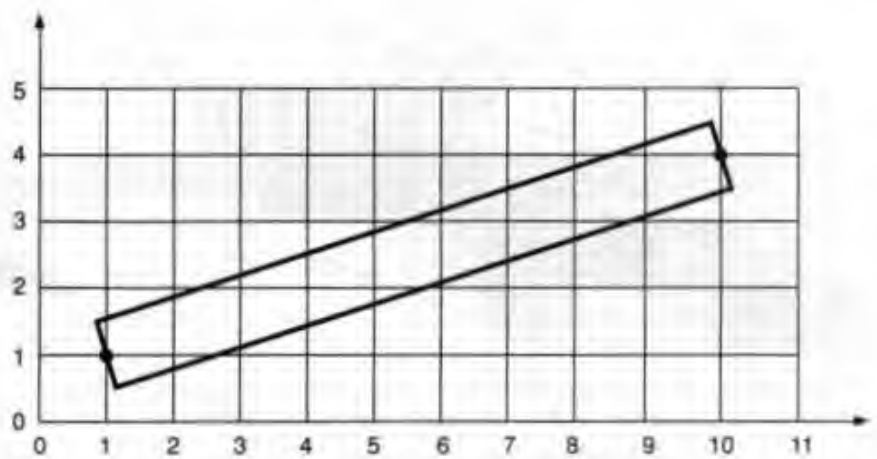


Figure 3.55 Line of nonzero width from point (1,1) to point (10,4).

Samsung/Google, EX1027, p. 152

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

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

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

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

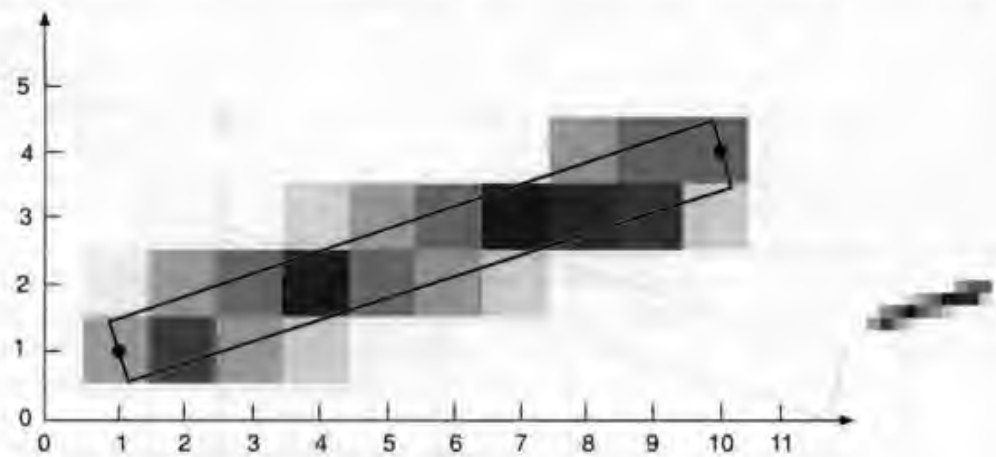


Fig. 3.56 Intensity proportional to area covered.

Samsung/Google, EX1027, p. 153

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

3.17.3 Weighted Area Sampling

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

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

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

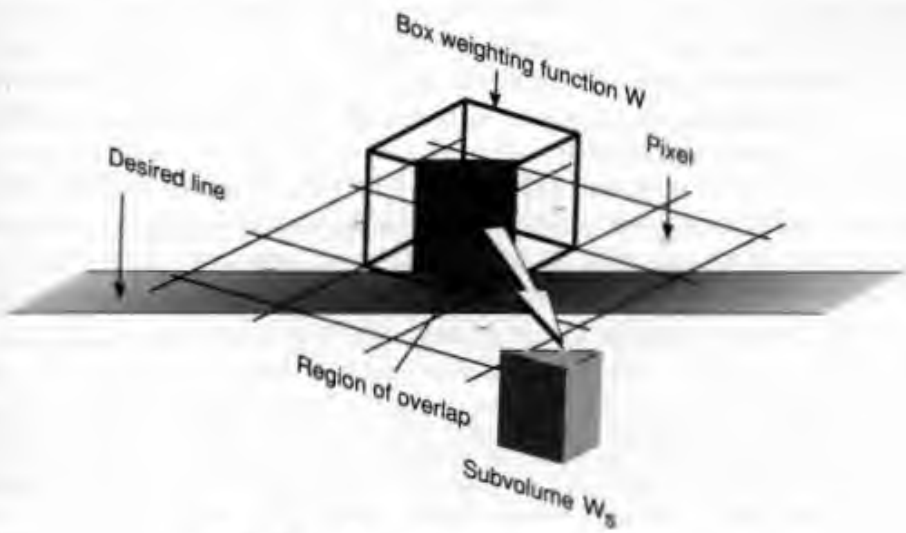


Fig. 3.57 Box filter for square pixel.

Samsung/Google, EX1027, p. 154

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

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

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

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

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

Samsung/Google, EX1027, p. 155

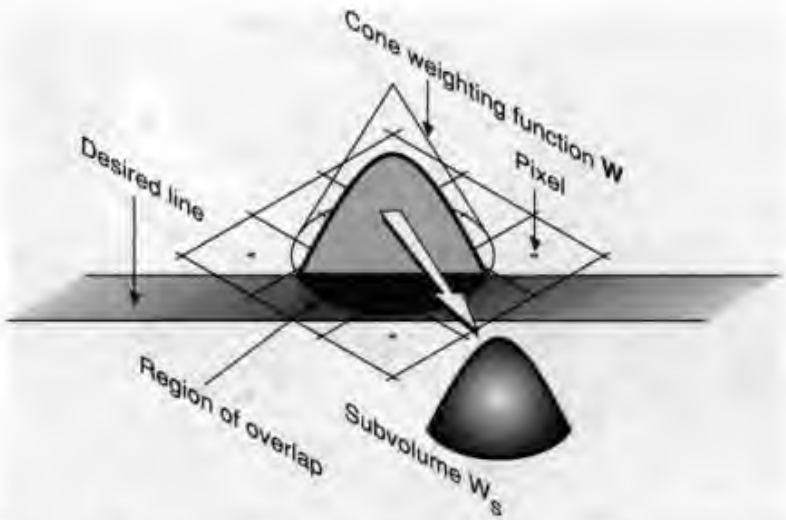


Fig. 3.58 Cone filter for circular pixel with diameter of two grid units.

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

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

3.17.4 Gupta–Sproull Antialiased Lines

The Gupta–Sproull scan-conversion algorithm for lines [GUPT81a] described in this section precomputes the subvolume of a normalized filter function defined by lines at various directed distances from the pixel center, and stores them in a table. We use a pixel area with radius equal to a grid unit—that is, to the distance between adjacent pixel centers—so that a line of unit thickness with slope less than 1 typically intersects three supports in a column, minimally two and maximally five, as shown in Fig. 3.60. For a radius of 1, each circle partially covers the circles of its neighboring pixels.

Figure 3.61 shows the geometry of the overlap between line and pixel that is used for table lookup of a function Filter (D, t). Here D is the (angle-independent) distance between pixel and line centers, t is a constant for lines of a given thickness, and function Filter() is dependent on the shape of the filter function. Gupta and Sproull's paper gives the table for a cone filter for a 4-bit display; it contains fractional values of Filter (D, t) for equal

Samsung/Google, EX1027, p. 156



Fig. 3.59 Filtered line drawing. The left half is unfiltered; the right half is filtered. (Courtesy of Branko Gerovac, Digital Equipment Corporation.)

increments of D ranging from 0 to 1.5 and for $t = 1$. The function, by definition, is 0 outside the support, for $D \geq 1 + \frac{1}{2} = \frac{24}{16}$ in this case. The precision of the distance is only 1 in 16, because it need not be greater than that of the intensity—4 bits, in this case.

Now we are ready to modify the midpoint line scan-conversion algorithm. As before, we use the decision variable d to choose between E and NE pixels, but must then set the intensity of the chosen pixel and its two vertical neighbors, on the basis of the distances from these pixels to the line. Figure 3.61 shows the relevant geometry; we can calculate the

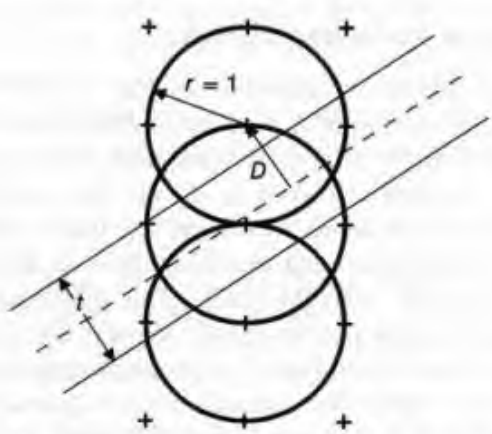


Fig. 3.60 One-unit-thick line intersects 3-pixel supports.

Samsung/Google, EX1027, p. 157

true, perpendicular distance D from the vertical distance v, using simple trigonometry.

Using similar triangles and knowing that the slope of the line is dy/dx, we can see from the diagram that

$$D = v \cos\phi = \frac{vdx}{\sqrt{dx^2 + dy^2}} \tag{3.2}$$

The vertical distance v between a point on the line and the chosen pixel with the same x coordinate is just the difference between their y coordinates. It is important to note that this distance is a signed value. That is, if the line passes below the chosen pixel, v is negative; if it passes above the chosen pixel, v is positive. We should therefore pass the absolute value of the distance to the filter function. The chosen pixel is also the middle of the 3 pixels that must be intensified. The pixel above the chosen one is a vertical distance $1 - v$ from the line, whereas the pixel below is a vertical distance $1 + v$ from the line. You may want to verify that these distances are valid regardless of the relative position of the line and the pixels, because the distance v is a signed quantity.

Rather than computing v directly, our strategy is to use the incremental computation of $d = F(M) = F(x_P + 1, y_P + \frac{1}{2})$. In general, if we know the x coordinate of a point on the line, we can compute that point's y coordinate using the relation developed in Section 3.2.2, $F(x, y) = 2(ax + by + c) = 0$:

$$y = (ax + c)/-b.$$

For pixel E, $x = x_P + 1$, and $y = y_P$, and $v = y - y_P$; thus

$$v = ((a(x_P + 1) + c)/-b) - y_P.$$

Now multiplying both sides by $-b$ and collecting terms,

$$-bv = a(x_P + 1) + by_P + c = F(x_P + 1, y_P)/2.$$

But $b = -dx$. Therefore, $vdx = F(x_P + 1, y_P)/2$. Note that vdx is the numerator of Eq. (3.2) for D, and that the denominator is a constant that can be precomputed. Therefore, we

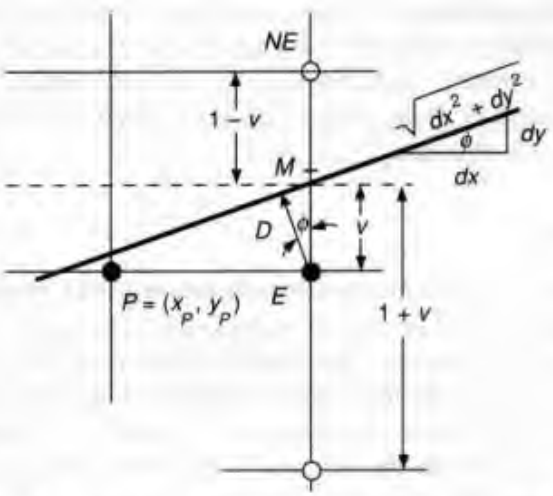


Fig. 3.61 Calculating distances to line in midpoint algorithm.

Samsung/Google, EX1027, p. 158

would like to compute vdx incrementally from the prior computation of $d = F(M)$, and avoid division by 2 to preserve integer arithmetic. Thus, for the pixel E,

$$\begin{aligned} 2vdx = F(x_P + 1, y_P) &= 2a(x_P + 1) + 2by_P + 2c \\ &= 2a(x_P + 1) + 2b(y_P + \tfrac{1}{2}) - 2b/2 + 2c \\ &= d + dx. \end{aligned}$$

Thus

$$D = \frac{d + dx}{2\sqrt{x^2 + y^2}}$$

and the constant denominator is $1/(2\sqrt{x^2 + y^2})$. The corresponding numerators for the pixels at $y_P + 1$ and $y_P - 1$ are then easily obtained as $2(1 - v)dx = 2dx - 2vdx$, and $2(1 + v)dx = 2dx + 2vdx$, respectively.

Similarly, for pixel NE,

$$\begin{aligned} 2vdx = F(x_P + 1, y_P + 1) &= 2a(x_P + 1) + 2b(y_P + \tfrac{1}{2}) + 2b/2 + 2c, \\ &= d - dx, \end{aligned}$$

and the corresponding numerators for the pixels at $y_P + 2$ and y_P are again $2(1 - v)dx = 2dx - 2vdx$ and $2(1 + v)dx = 2dx + 2vdx$, respectively.

We have put a dot in front of the statements added to the midpoint algorithm of Section 3.2.2 to create the revised midpoint algorithm shown in Fig. 3.62. The WritePixel of E or NE has been replaced by a call to IntensifyPixel for the chosen pixel and its vertical neighbors; IntensifyPixel does the table lookup that converts the absolute value of the distance to weighted area overlap, a fraction of maximum intensity. In an actual implementation, this simple code, of course, would be inline.

The Gupta–Sproull algorithm provides an efficient incremental method for antialiasing lines, although fractional arithmetic is used in this version. The extension to antialiasing endpoints by using a separate look-up table is covered in Section 19.3.5. Since the lookup works for the intersection with the edge of the line, it can also be used for the edge of an arbitrary polygon. The disadvantage is that a single look-up table applies to lines of a given thickness only. Section 19.3.1 discusses more general techniques that consider any line as two parallel edges an arbitrary distance apart. We also can antialias characters either by filtering them (see Section 19.4) or, more crudely, by taking their scanned-in bitmaps and manually softening pixels at edges.

3.18 SUMMARY

In this chapter, we have taken our first look at the fundamental clipping and scan-conversion algorithms that are the meat and potatoes of raster graphics packages. We have covered only the basics here; many elaborations and special cases must be considered for robust implementations. Chapter 19 discusses some of these, as well as such topics as general regions and region filling. Other algorithms that operate on bitmaps or pixmaps are discussed in Chapter 17, and a fuller treatment of the theory and practice of antialiasing is found in Chapters 14 and 19.

Samsung/Google, EX1027, p. 159

```
static void IntensifyPixel (int, int, double)

void AntiAliasedLineMidpoint (int x0, int y0, int x1, int y1)
/* This algorithm uses Gupta-Sproull's table of intensity as a function of area */
/* coverage for a circular support in the IntensifyPixel function. Note that */
/* overflow may occur in the computation of the denominator for 16-bit integers, */
/* because of the squares. */
{
    int dx = x1 - x0;
    int dy = y1 - y0;
    int d = 2 * dy - dx;                          /* Initial value d_start as before */
    int incrE = 2 * dy;                           /* Increment used for move to E */
    int incrNE = 2 * (dy - dx);                   /* Increment used for move to NE */
•   int two_v_dx = 0;                             /* Numerator; v = 0 for start pixel */
•   double invDenom = 1.0 /
        (2.0 * sqrt (dx * dx + dy * dy));         /* Precomputed inverse denominator */
•   double two_dx_invDenom =                      /* Precomputed constant */
        2.0 * dx * invDenom;
    int x = x0;
    int y = y0;
•   IntensifyPixel (x, y, 0);                     /* Start pixel */
•   IntensifyPixel (x, y + 1, two_dx_invDenom);   /* Neighbor */
•   IntensifyPixel (x, y - 1, two_dx_invDenom);   /* Neighbor */
    while (x < x1) {
        if (d < 0) {                              /* Choose E */
•           two_v_dx = d + dx;
            d += incrE;
            x++;
        } else {                                  /* Choose NE */
•           two_v_dx = d - dx;
            d += incrNE;
            x++;
            y++;
        }
        /* Now set chosen pixel and its neighbors */
•       IntensifyPixel (x, y, two_v_dx * invDenom);
•       IntensifyPixel (x, y + 1, two_v_dx_invDenom - two_v_dx * invDenom);
•       IntensifyPixel (x, y - 1, two_v_dx_invDenom + two_v_dx * invDenom);
    }
}   /* AntiAliasedLineMidpoint */

void IntensifyPixel (int x, int y, double distance)
{
    double intensity = Filter (Round (fabs (distance)));
    /* Table lookup done on an integer index; thickness 1 */
    WritePixel (x, y, intensity);
}   /* IntensifyPixel */
```

Fig. 3.62 Gupta–Sproull algorithm for antialiased scan conversion of lines.

Samsung/Google, EX1027, p. 160

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

EXERCISES

3.1 Implement the special-case code for scan converting horizontal and vertical lines, and lines with slopes of ± 1.

3.2 Modify the midpoint algorithm for scan converting lines (Fig. 3.8) to handle lines at any angle.

3.3 Show why the point-to-line error is always $\leq \frac{1}{2}$ for the midpoint line scan-conversion algorithm.

3.4 Modify the midpoint algorithm for scan converting lines of Exercise 3.2 to handle endpoint order and intersections with clip edges, as discussed in Section 3.2.3.

3.5 Modify the midpoint algorithm for scan converting lines (Exercise 3.2) to write pixels with varying intensity as a function of line slope.

3.6 Modify the midpoint algorithm for scan converting lines (Exercise 3.2) to deal with endpoints that do not have integer coordinates—this is easiest if you use floating point throughout your algorithm. As a more difficult exercise, handle lines of *rational* endpoints using only integers.

3.7 Determine whether the midpoint algorithm for scan converting lines (Exercise 3.2) can take advantage of symmetry by using the decision variable d to draw simultaneously from both ends of the line toward the center. Does your algorithm consistently accommodate the case of equal error on an arbitrary choice that arises when dx and dy have a largest common factor c and dx/c is even and dy/c is odd ($0 < dy < dx$), as in the line between (0, 0) and (24, 9)? Does it deal with the subset case in which dx is an integer multiple of $2dy$, such as for the line between (0, 0) and (16, 4)? (Contributed by J. Bresenham.)

3.8 Show how polylines may share more than vertex pixels. Develop an algorithm that avoids writing pixels twice. Hint: Consider scan conversion and writing to the canvas in **xor** mode as separate phases.

3.9 Expand the pseudocode for midpoint ellipse scan conversion of Fig. 3.21 to code that tests properly for various conditions that may arise.

3.10 Apply the technique of forward differencing shown for circles in Section 3.3.2 to develop the second-order forward differences for scan converting standard ellipses. Write the code that implements this technique.

3.11 Develop an alternative to the midpoint circle scan-conversion algorithm of Section 3.3.2 based on a piecewise-linear approximation of the circle with a polyline.

3.12 Develop an algorithm for scan converting unfilled rounded rectangles with a specified radius for the quarter-circle corners.

Samsung/Google, EX1027, p. 161

3.13 Write a scan-conversion procedure for solidly filled upright rectangles at arbitrary screen positions that writes a bilevel frame buffer efficiently, an entire word of pixels at a time.

3.14 Construct examples of pixels that are "missing" or written multiple times, using the rules of Section 3.6. Try to develop alternative, possibly more complex, rules that do not draw shared pixels on shared edges twice, yet do not cause pixels to be missing. Are these rules worth the added overhead?

3.15 Implement the pseudocode of Section 3.6 for polygon scan conversion, taking into account in the span bookkeeping of potential sliver polygons.

3.16 Develop scan-conversion algorithms for triangles and trapezoids that take advantage of the simple nature of these shapes. Such algorithms are common in hardware.

3.17 Investigate triangulation algorithms for decomposing an arbitrary, possibly concave or self-intersecting, polygon into a mesh of triangles whose vertices are shared. Does it help to restrict the polygon to being, at worse, concave without self-intersections or interior holes? (See also [PREP85].)

3.18 Extend the midpoint algorithm for scan converting circles (Fig. 3.16) to handle filled circles and circular wedges (for pie charts), using span tables.

3.19 Extend the midpoint algorithm for scan converting ellipses (Fig. 3.21) to handle filled elliptical wedges, using span tables.

3.20 Implement both absolute and relative anchor algorithms for polygon pattern filling, discussed in Section 3.9, and contrast them in terms of visual effect and computational efficiency.

3.21 Apply the technique of Fig. 3.30 for writing characters filled with patterns in opaque mode. Show how having a copyPixel with a write mask may be used to good advantage for this class of problems.

3.22 Implement a technique for drawing various symbols such as cursor icons represented by small bitmaps so that they can be seen regardless of the background on which they are written. Hint: Define a mask for each symbol that "encloses" the symbol—that is, that covers more pixels than the symbol—and that draws masks and symbols in separate passes.

3.23 Implement thick-line algorithms using the techniques listed in Section 3.9. Contrast their efficiency and the quality of the results they produced.

3.24 Extend the midpoint algorithm for scan converting circles (Fig. 3.16) to handle thick circles.

3.25 Implement a thick-line algorithm that accommodates line style as well as pen style and pattern.

3.26 Implement scissoring as part of scan converting lines and unfilled polygons, using the fast-scan-plus-backtracking technique of checking every *i*th pixel. Apply the technique to filled and thick lines and to filled polygons. For these primitives, contrast the efficiency of this type of on-the-fly clipping with that of analytical clipping.

3.27 Implement scissoring as part of scan converting unfilled and filled circles and ellipses. For these primitives, contrast the feasibility and efficiency of this type of on-the-fly clipping with that of analytical clipping.

3.28 Modify the Cohen–Sutherland line-clipping algorithm of Fig. 3.41 to avoid recalculation of slopes during successive passes.

3.29 Contrast the efficiency of the Sutherland–Cohen and Cyrus–Beck algorithms for several typical and atypical cases, using instruction counting. Are horizontal and vertical lines handled optimally?

3.30 Consider a convex polygon with n vertices being clipped against a clip rectangle. What is the maximum number of vertices in the resulting clipped polygon? What is the minimum number? Consider the same problem for a concave polygon. How many polygons might result? If a single polygon results, what is the largest number of vertices it might have?

Samsung/Google, EX1027, p. 162

3.31 Explain why the Sutherland–Hodgman polygon-clipping algorithm works for only convex clipping regions.

3.32 Devise a strategy for subdividing a pixel and counting the number of subpixels covered (at least to a significant degree) by a line, as part of a line-drawing algorithm using unweighted area sampling.

3.33 Create tables with various decreasing functions of the distance between pixel center and line center. Use them in the antialiased line algorithm of Fig. 3.62. Contrast the results produced with those produced by a box-filtered line.

3.34 Generalize the antialiasing techniques for lines to polygons. How might you handle nonpolygonal boundaries of curved primitives and characters?

Samsung/Google, EX1027, p. 163

4
Graphics Hardware

In this chapter, we describe how the important hardware elements of a computer graphics display system work. Section 4.1 covers hardcopy technologies: printers, pen plotters, electrostatic plotters, laser printers, ink-jet plotters, thermal-transfer plotters, and film recorders. The basic technological concepts behind each type of device are described briefly, and a concluding section compares the various devices. Section 4.2, on display technologies, discusses monochrome and color shadow-mask CRTs, the direct-view storage tube (DVST), liquid-crystal displays (LCDs), plasma panels, electroluminescent displays, and several more specialized technologies. Again, a concluding section discusses the pros and cons of the various display technologies.

Raster display systems, which can use any of the display technologies discussed here, are discussed in Section 4.3. A simple, straightforward raster system is first introduced, and is then enhanced with respect to graphics functionality and integration of raster- and general-purpose processors into the system address space. Section 4.4 describes the role of the look-up table and video controller in image display, color control, animation, and image mixing. The almost-obsolete vector (also called *random*, *calligraphic*, *stroke*) display system is discussed briefly in Section 4.5, followed in Section 4.6 by user interaction devices such as tablets, mice, touch panels, and so on. Again, operational concepts rather than technological details are stressed. Section 4.7 briefly treats image-input devices, such as film scanners, by means of which an existing image can by input to a computer.

Figure 4.1 shows the relation of these hardware devices to one another. The key element is the integrated CPU and display processor known as a *graphics workstation*, typically consisting of a CPU capable of executing at least several million instructions per second (MIPS), a large disk, and a display with resolution of at least 1000 by 800. The

Samsung/Google, EX1027, p. 164

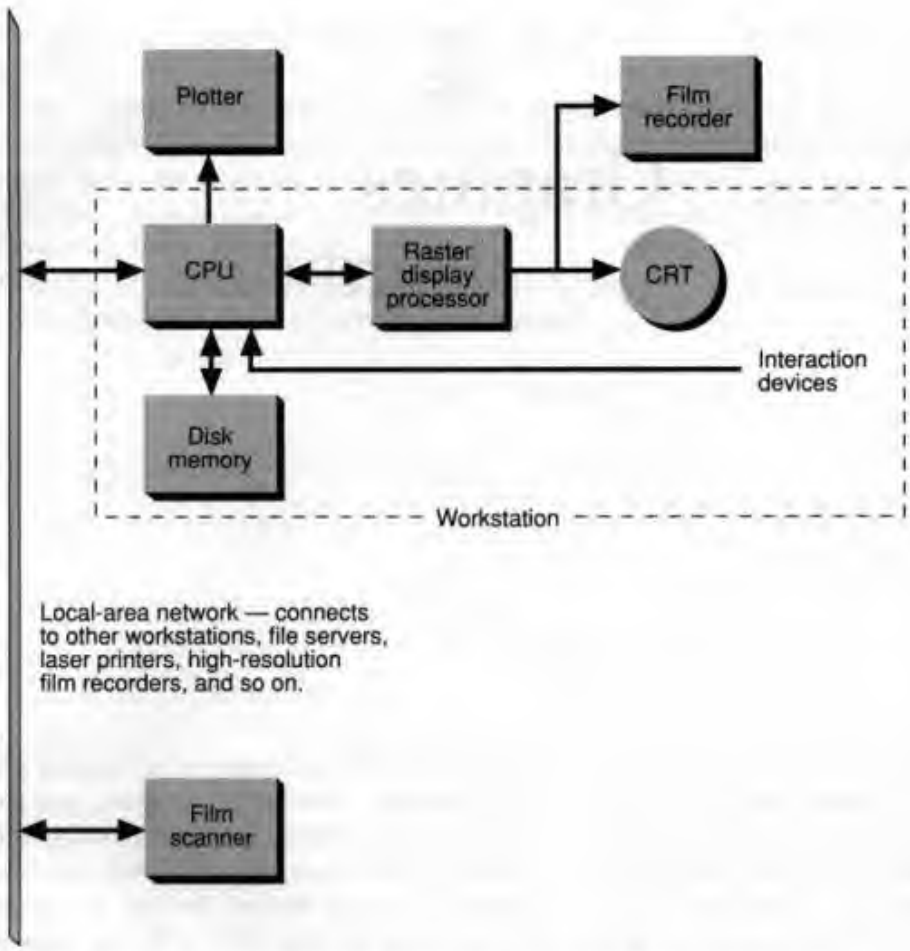


Fig. 4.1 Components of a typical interactive graphics system.

local-area network connects multiple workstations for file sharing, electronic mail, and access to shared peripherals such as high-quality film plotters, large disks, gateways to other networks, and higher-performance computers.

4.1 HARDCOPY TECHNOLOGIES

In this section, we discuss various hardcopy technologies, then summarize their characteristics. Several important terms must be defined first.

The image quality achievable with display devices depends on both the addressability and the dot size of the device. *Dot size* (also called *spot size*) is the diameter of a single dot on the device's output. *Addressability* is the number of individual (not necessarily distinguishable) dots per inch that can be created; it may differ in the horizontal and vertical directions. Addressability in x is just the reciprocal of the distance between the centers of dots at addresses (x, y) and $(x + 1, y)$; addressability in y is defined similarly. *Interdot distance* is the reciprocal of addressability.

Samsung/Google, EX1027, p. 165

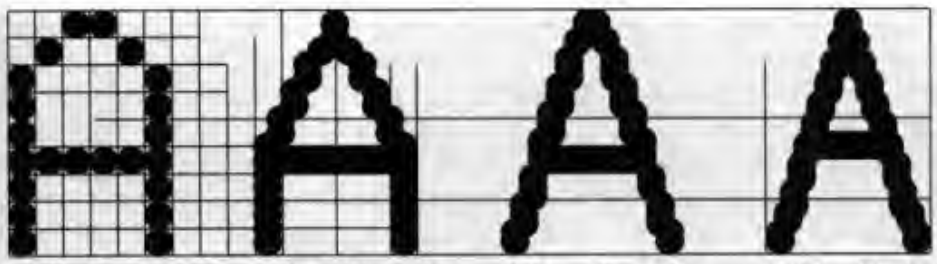


Fig. 4.2 The effects of various ratios of the dot size to the interdot distance.

It is usually desirable that dot size be somewhat greater than the interdot distance, so that smooth shapes can be created. Figure 4.2 illustrates this reasoning. Tradeoffs arise here, however: dot size several times the interdot distance allows very smooth shapes to be printed, whereas a smaller dot size allows finer detail.

Resolution, which is related to dot size and can be no greater than addressability, is the number of distinguishable lines per inch that a device can create. Resolution is defined as the closest spacing at which adjacent black and white lines can be distinguished by observers (this again implies that horizontal and vertical resolution may differ). If 40 black lines interleaved with 40 white lines can be distinguished across one inch, the resolution is 80 lines per inch (also referred to as 40 line-pairs per inch).

Resolution also depends on the cross-sectional intensity distribution of a spot. A spot with sharply delineated edges yields higher resolution than does one with edges that trail off, as shown in Fig. 4.3.

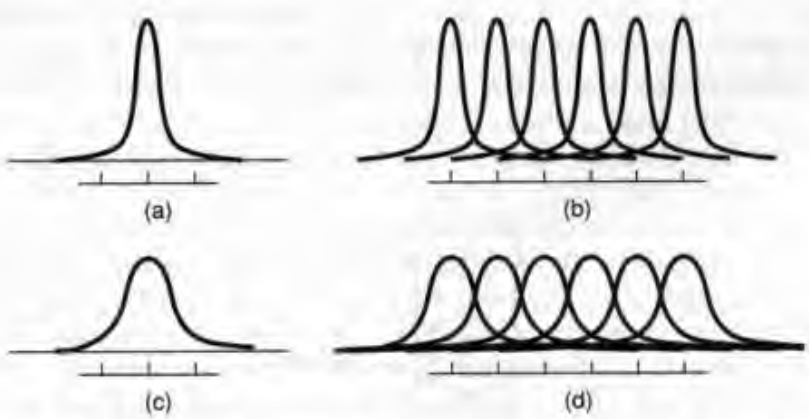


Fig. 4.3 The effect of cross-sectional spot intensity on resolution. (a) A spot with well-defined edges. (b) Several such overlapping spots. (c) A wider spot, with less height, since the energy is spread out over a larger area; its edges are not well defined, as are those in (a). (d) Several of these spots overlapping. The distinction between the peaks in (b) is much clearer in (d). The actual image intensity is the sum of each spot's intensity.

Samsung/Google, EX1027, p. 166

Many of the devices to be discussed can create only a few colors at any one point. Additional colors can be obtained with dither patterns, described in Chapter 13, at the cost of decreased spatial resolution of the resulting image.

Dot-matrix printers use a print head of from 7 to 24 *pins* (thin, stiff pieces of wire), each of which can be individually *fired*, to strike a ribbon against the paper. The print head moves across the paper one step at a time, the paper is advanced one line, and the print head makes another pass across the paper. Hence, these printers are raster output devices, requiring scan conversion of vector images prior to printing.

The addressability of a dot-matrix printer does not need to be limited by the physical distance between pins on the print head. There can be two columns of pins, offset vertically by one-half the interpin spacing, as seen in Fig. 4.4. Alternatively, two passes over the paper can be used to achieve the same effect, by advancing the paper by one-half the interpin spacing between the first and second passes.

Colored ribbons can be used to produce color hardcopy. Two approaches are possible. The first is using multiple print heads, each head with a different color ribbon. Alternatively and more commonly, a single print head is used with a multicolored ribbon.

More colors than are actually on the ribbon can be created by overstriking two different colors at the same dot on the paper. The color on top may be somewhat stronger than that underneath. Up to eight colors can be created at any one dot by overstriking with three colors—typically cyan, magenta, and yellow. However, the black resulting from striking all three is quite muddy, so a true black is often added to the ribbon.

Just as there are random and raster displays, so too there are random and raster plotters. *Pen plotters* move a pen over a piece of paper in random, vector-drawing style. In drawing a line, the pen is positioned at the start of the line, lowered to the paper, moved in a straight path to the endpoint of the line, raised, and moved to the start of the next line. There are two basic varieties of pen plotters. The *flatbed plotter* moves the pen in x and y on a sheet of paper spread out on a table and held down by electrostatic charge, by vacuum, or simply by being stretched tightly (Fig. 4.5). A carriage moves longitudinally over the table. On the carriage is a pen mount moving latitudinally along the carriage; the pen can be raised and lowered. Flatbed plotters are available in sizes from 12 by 18 inches to 6 by 10 feet and

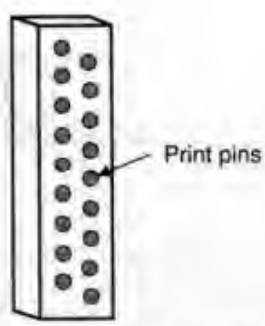


Fig. 4.4 A dot-matrix print head with two columns of print pins, offset vertically by half the interpin spacing to increase resolution.

Samsung/Google, EX1027, p. 167



Fig. 4.5 A flatbed plotter. (Courtesy of CalComp–California Computer Products, Inc.)

larger. In some cases, the "pen" is a light source for exposing photographic negatives or a knife blade for scribing, and often pens of multiple colors or widths are used.

In contrast, *drum plotters* move the paper along one axis and the pen along the other axis. Typically, the paper is stretched tightly across a drum, as shown in Fig. 4.6. Pins on the drum engage prepunched holes in the paper to prevent slipping. The drum can rotate both forward and backward. By contrast, many *desk-top plotters* move the paper back and forth between pinch rollers, while the pen moves across the paper (Fig. 4.7).

Pen plotters include a microprocessor that accepts commands such as "draw line," "move," "draw circle," "draw text," "set line style," and "select pen." (The microprocessor sometimes also optimizes pen movements to minimize the distance the pen moves while up; Anderson [ANDE83] has developed an efficient algorithm for doing this.) The microprocessor decomposes the output primitives into incremental pen movements in any of the eight principal directions. A feedback system of position sensors and servomotors implements the motion commands, and an electromagnet raises and lowers the pen. Plotting speed depends on the acceleration and velocity with which the pen can be moved. In turn, pen acceleration is partially a function of the mass of the plot head; many multipen plotters keep all but the active pens at the side of the plotter to minimize this mass.

In contrast to the pen plotter, the *electrostatic plotter* places a negative charge on those parts of white paper that are to be black, then flows positively charged black toner over the paper (Fig. 4.8). The toner particles adhere to the paper where the charge has been deposited. The charge is placed on the paper, which in current systems can be up to 72

Samsung/Google, EX1027, p. 168

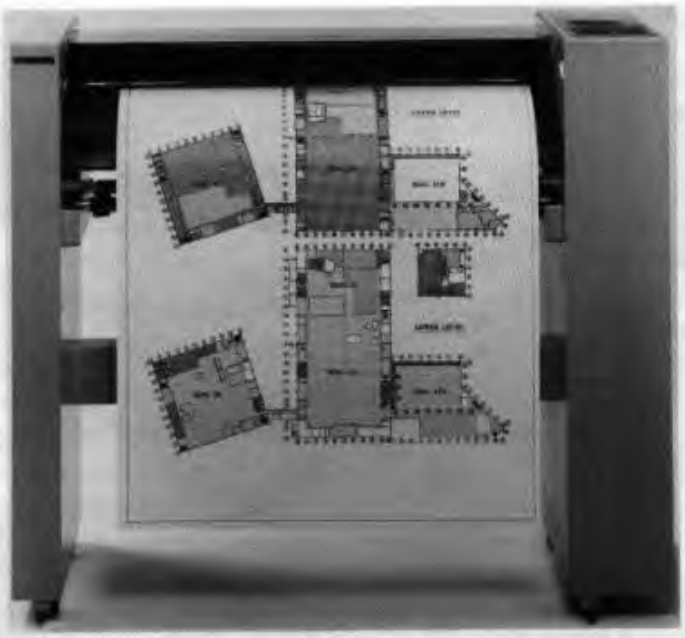


Fig. 4.6 A drum plotter. (Courtesy of Hewlett-Packard Company.)

inches wide, one row at a time. The paper moves at speeds up to 3 inches per second under a fine comb of electric contacts spaced horizontally 100 to 400 to the inch. Each contact is either on (to impart a negative charge) or off (to impart no charge). Each dot on an electrostatic plot is either black or white; gray levels must be created with dither patterns.



Fig. 4.7 A desk-top plotter. (Courtesy of Hewlett-Packard Company.)

Samsung/Google, EX1027, p. 169

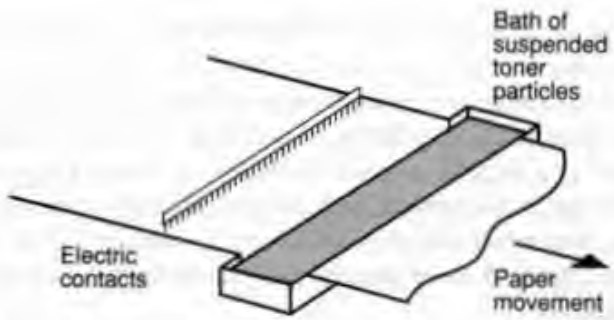


Fig. 4.8 Organization of an electrostatic plotter.

Electrostatic plotters may include a scan-conversion capability, or scan conversion can be done by the CPU. In the latter case, because the density of information on a 400 by 400 dot-per-square inch electrostatic plotter is quite high (see Exercise 4.1), correspondingly high transfer rates are needed.

Some color electrostatic plotters make multiple passes over the paper, rewinding to the start of the plot after each pass. On the first pass, black calibration marks are placed near the edge of the paper. Subsequent passes complete the plot with black, cyan, magenta, and yellow toners, using the calibration marks to maintain alignment. Others use multiple heads to deposit all the colors in a single pass.

Electrostatic plotters are often faster than pen plotters, and can also double as high-quality printers. On the other hand, pen plotters create images with higher contrast than those made by electrostatic plotters, since the latter deposit a toner even in areas where the paper is not negatively charged.

Laser printers scan a laser beam across a positively charged rotating drum coated with selenium. The areas hit by the laser beam lose their charge, and the positive charge remains only where the copy is to be black. A negatively charged powdered toner adheres to the positive areas of the drum and is then transferred to blank paper to form the copy. In color xerography, this process is repeated three times, once for each primary color. Figure 4.9 is a partial schematic of a monochrome laser printer.

Just as with the electrostatic plotter, the positive charge is either present or not present at any one spot on the drum, and there is either black or not black at the corresponding spot

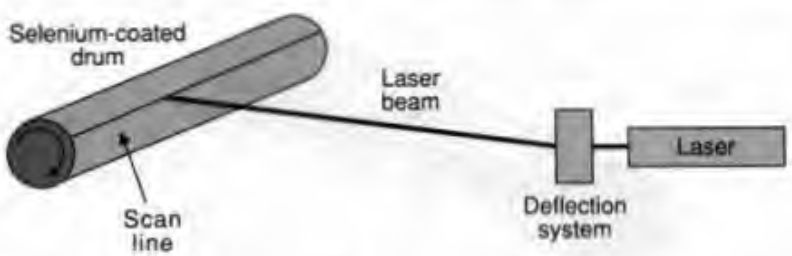


Fig. 4.9 Organization of a laser printer (the toner-application mechanism and the paper feeder are not shown).

Samsung/Google, EX1027, p. 170

on the copy. Hence, the laser printer is a two-level monochrome device or an eight-color color device.

Laser printers have a microprocessor to do scan conversion and to control the printer. An increasing number of laser printers accept the Postscript document and image description language as a defacto standard [ADOB85a]. Postscript provides a procedural description of an image to be printed, and can also be used to store image descriptions (Chapter 19). Most laser printers work with 8.5- by 11-inch or 8.5- by 14-inch paper, but considerably wider (30-inch) laser printers are available for engineering drawing and map-making applications.

Ink-jet printers spray cyan, magenta, yellow, and sometimes black ink onto paper. In most cases, the ink jets are mounted on a head in a printerlike mechanism. The print head moves across the page to draw one scan line, returns while the paper advances by one inter–scan-line spacing, and draws the next scan line. Slight irregularities in interline spacing can arise if the paper transport moves a bit too much or too little. Another approach is to wrap the paper around a drum; the drum then rotates rapidly while the print head slowly moves along the drum. Figure 4.10 shows this arrangement. In both cases, all the colors are deposited simultaneously, unlike the multipass laser and electrostatic plotters and printers. Most ink-jet printers are limited to on–off (i.e., bilevel) control of each pixel: a few have a variable dot-size capability.

Some ink-jet printers accept video input as well as digital, which makes them attractive for creating hardcopy images of raster display screens. Note that the resolution of a resulting image is limited by the resolution of the video input—typically between 640 by 480 and 1280 by 1024. Ink-jet printers tend to require more maintenance than do many of the other types.

Thermal-transfer printers, another raster hardcopy device, are reminiscent of electrostatic plotters. Finely spaced (typically 200-per-inch) heating nibs transfer pigments from colored wax paper to plain paper. The wax paper and plain paper are drawn together over the strip of heating nibs, which are selectively heated to cause the pigment transfer. For color printing (the most common use of this technology), the wax paper is on a roll of alternating cyan, magenta, yellow, and black strips, each of a length equal to the paper size. Because the nibs heat and cool very rapidly, a single color hardcopy image can be created in less than 1 minute. Some thermal-transfer printers accept a video signal and digital bitmap input, making them convenient for creating hardcopy of video images.

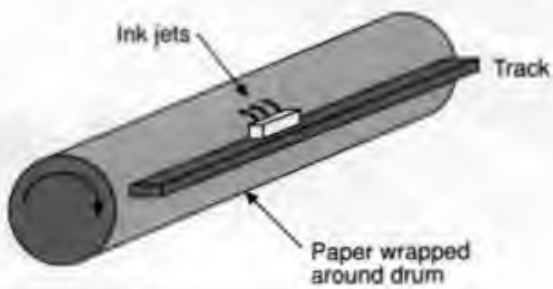


Fig. 4.10 A rotary ink-jet plotter.

Samsung/Google, EX1027, p. 171

Thermal sublimation dye transfer printers work similarly to the thermal transfer printers, except the heating and dye transfer process permit 256 intensities each of cyan, magenta, and yellow to be transferred, creating high-quality full-color images with a spatial resolution of 200 dots per inch. The process is slower than wax transfer, but the quality is near-photographic.

A *camera* that photographs an image displayed on a cathode-ray (television) tube can be considered another hardcopy device. This is the most common hardcopy technology we discuss that yields a large number of colors at a single resolution point; film can capture many different colors.

There are two basic techniques for color film recorders. In one, the camera records the color image directly from a color CRT. Image resolution is limited because of the shadow mask of the color monitor (see Section 4.2.2) and the need to use a raster scan with the color monitor. In the other approach, a black-and-white CRT is photographed through color filters, and the different color components of the image are displayed in sequence (Fig 4.11). This technique yields very high-quality raster or vector images. Colors are mixed by double-exposing parts of the image through two or more filters, usually with different CRT intensities.

Input to film recorders can be a raster video signal, a bitmap, or vector-style instructions. Either the video signal can drive a color CRT directly, or the red, green, and blue components of the signal can be electronically separated for time-sequential display through filters. In either case, the video signal must stay constant during the entire recording cycle, which can be up to 1 minute if relatively slow (low-sensitivity) film is being used. High-speed, high-resolution bitmap or vector systems are expensive, because the drive electronics and CRT itself must be designed and calibrated carefully. As speed and resolution decrease, costs are reduced dramatically.

The recently developed *Cycolor* technique embeds in paper millions of microcapsules filled with one of three colored dyes—cyan, magenta, or yellow. The capsules harden selectively when exposed to light of a specific color. For instance, when exposed to green

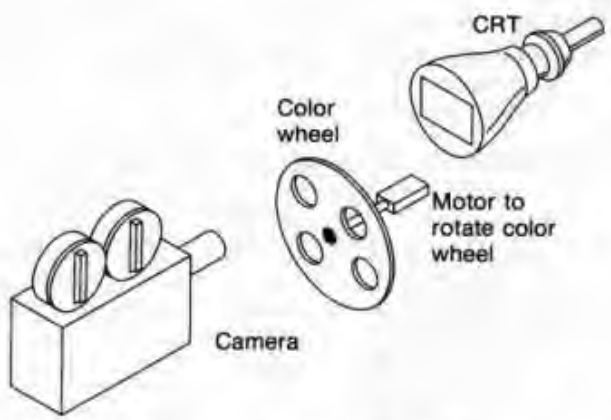


Fig. 4.11 A film recorder for making color photographs using colored filters.

Samsung/Google, EX1027, p. 172

TABLE 4.1 A COMPARISON OF SEVERAL MONOCHROME HARDCOPY TECHNOLOGIES*

	Pen plotter	Dot matrix	Electro-static	Laser	Ink jet	Thermal	Photo
intensity levels per dot	2	2	2	2	2–many	2	many
addressability, points per inch	1000+	to 250	to 400	to 1500	to 200	to 200	to 800
dot size, thousandths of inch	6–15	10–18	8	5	8–20	7–10	6–20
relative cost range	L–M	VL–L	M	M–H	L–M	L–M	L–H
relative cost per image	L	VL	M	M	L	M	H
image quality	L–M	L–M	M	H	M	M	H
speed	L	M	H–H	M–H	M	M	L

*VL = very low, L = low, M = medium, H = high.

light, chemicals in the magenta-filled capsule cause that capsule to harden. The paper is passed through pressure rollers and pressed against a sheet of plain paper. The unhardened capsules (cyan and yellow, in this example) break, but the hardened capsule (magenta) does not. The cyan and yellow colors mix and are transferred to the plain paper, creating a high-quality green image. Unlike most other technologies, this one requires only a single pass.

Table 4.1 summarizes the differences among black-and-white hardcopy devices; Table 4.2 covers most of the color hardcopy devices. Considerable detail on the technology of hardcopy devices can be found in [DURB88]. The current pace of technological innovation is, of course, so great that the relative advantages and disadvantages of some of these devices will surely change. Also, some of the technologies are available in a wide range of prices and performances. Film recorders and pen plotters, for instance, can cost from about $1000 to $100,000.

Note that, of all the color devices, only the film recorder, Cycolor, and some ink-jet printers can capture a wide range of colors. All the other technologies use essentially a binary on–off control for the three or four colors they can record directly. Note also that color control is tricky: there is no guarantee that the eight colors on one device will look anything like the eight colors on the display or on another hardcopy device (Chapter 13).

Wide laser printers are becoming available: they will slowly preempt electrostatic plotters, which have a lower contrast ratio between black and white and are more difficult to maintain.

TABLE 4.2 A COMPARISON OF SEVERAL COLOR HARDCOPY TECHNOLOGIES*

	Pen plotter	Dot matrix	Electro-static	Laser	Ink jet	Thermal	Photo
color levels per dot	to 16	8	8	8	8–many	8–many	many
addressability, points per inch	1000+	to 250	to 400	to 1500	to 200	to 200	to 800
dot size, thousandths of inch	15–6	18–10	8	5	20–8	10–7	20–6
relative cost range	L–M	VL	M–H	M–H	L–M	M	M–H
relative cost per image	L	VL	M	M	L	M	H
image quality	L–M	L	M	H	M	M–H	M–H
speed	L	L–M	M	M	M	L–M	L

*VL = very low, L = low, M = medium, H = high.

Samsung/Google, EX1027, p. 173

4.2 DISPLAY TECHNOLOGIES

Interactive computer graphics demands display devices whose images can be changed quickly. Nonpermanent image displays allow an image to be changed, making possible dynamic movement of portions of an image. The CRT is by far the most common display device and will remain so for many years. However, solid-state technologies are being developed that may, in the long term, substantially reduce the dominance of the CRT.

The *monochromatic CRTs* used for graphics displays are essentially the same as those used in black-and-white home television sets. Figure 4.12 shows a highly stylized cross-sectional view of a CRT. The electron gun emits a stream of electrons that is accelerated toward the phosphor-coated screen by a high positive voltage applied near the face of the tube. On the way to the screen, the electrons are forced into a narrow beam by the focusing mechanism and are directed toward a particular point on the screen by the magnetic field produced by the deflection coils. When the electrons hit the screen, the phosphor emits visible light. Because the phosphor's light output decays exponentially with time, the entire picture must be *refreshed* (redrawn) many times per second, so that the viewer sees what appears to be a constant, unflickering picture.

The refresh rate for raster-scan displays is usually at least 60 frames per second, and is independent of picture complexity. The refresh rate for vector systems depends directly on picture complexity (number of lines, points, and characters): The greater the complexity, the longer the time taken by a single refresh cycle and the lower the refresh rate.

The stream of electrons from the heated cathode is accelerated toward the phosphor by a high voltage, typically 15,000 to 20,000 volts, which determines the velocity achieved by the electrons before they hit the phosphor. The control-grid voltage determines how many electrons are actually in the electron beam. The more negative the control-grid voltage is, the fewer the electrons that pass through the grid. This phenomenon allows the spot's

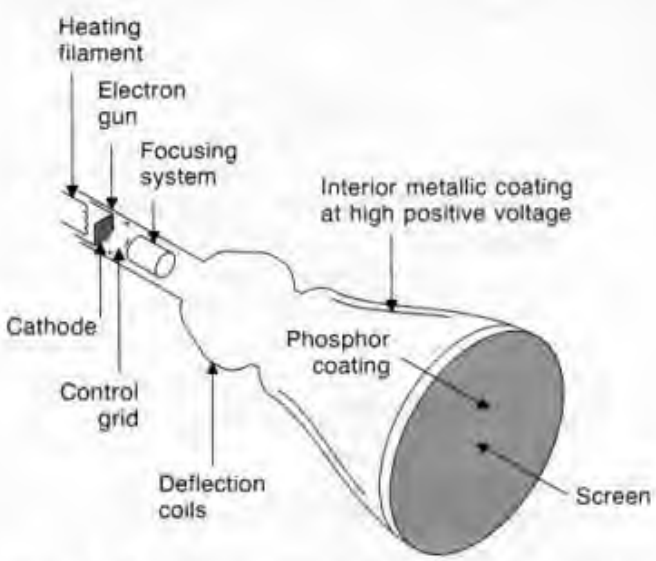


Fig. 4.12 Cross-section of a CRT (not to scale).

Samsung/Google, EX1027, p. 174

intensity to be controlled, because the light output of the phosphor decreases as the number of electrons in the beam decreases.

The focusing system concentrates the electron beam so that the beam converges to a small point when it hits the phosphor coating. It is not enough for the electrons in the beam to move parallel to one another. They would diverge because of electron repulsion, so the focusing system must make them converge to counteract the divergence. With the exception of this tendency to diverge, focusing an electron beam is analogous to focusing a light beam. An optical lens and an electron lens both have a focal distance, which in the case of the CRT is set such that the beam is focused on the screen. The cross-sectional electron density of the beam is Gaussian (that is, normal), and the intensity of the light spot created on the phosphor has the same distribution, as was shown in Fig. 4.3. The spot thus has no distinct edges, and hence spot size is usually specified as the diameter at which the intensity is 50 percent of that at the center of the spot. The typical spot size in a high-resolution monochrome CRT is 0.005 inches.

Figure 4.13 illustrates both the focusing system of and a difficulty with CRTs. The beam is shown in two positions. In one case, the beam converges at the point at which it strikes the screen. In the second case, however, the convergence point is behind the screen, and the resulting image is therefore somewhat blurred. Why has this happened? The faceplates of most CRTs are nearly flat, and hence have a radius of curvature far greater than the distance from the lens to the screen. Thus, not all points on the screen are equidistant from the lens, and if the beam is in focus when it is directed at the center of the screen, it is not in focus anywhere else on the screen. The further the beam is deflected from the center, the more defocused it is. In high-precision displays, the system solves this problem by focusing the lens dynamically as a function of the beam's position; making CRTs with sharply curved faceplates is not a good solution.

When the electron beam strikes the phosphor-coated screen of the CRT, the individual electrons are moving with kinetic energy proportional to the acceleration voltage. Some of this energy is dissipated as heat, but the rest is transferred to the electrons of the phosphor

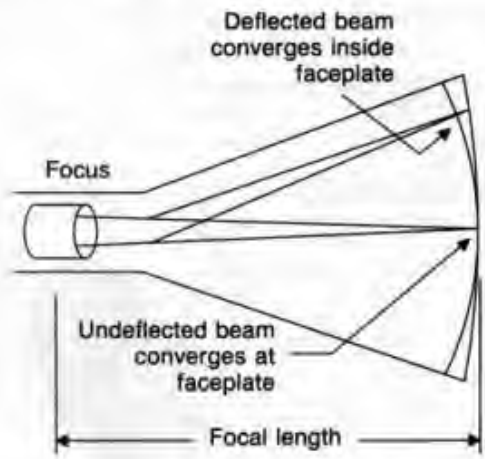


Fig. 4.13 Focusing of the electron beam. The focal length varies as a function of the deflection angle.

Samsung/Google, EX1027, p. 175

atoms, making them jump to higher quantum-energy levels. In returning to their previous quantum levels, these excited electrons give up their extra energy in the form of light, at frequencies (i.e., colors) predicted by quantum theory. Any given phosphor has several different quantum levels to which electrons can be excited, each corresponding to a color associated with the return to an unexcited state. Further, electrons on some levels are less stable and return to the unexcited state more rapidly than others. A phosphor's *fluorescence* is the light emitted as these very unstable electrons lose their excess energy while the phosphor is being struck by electrons. *Phosphorescence* is the light given off by the return of the relatively more stable excited electrons to their unexcited state once the electron beam excitation is removed. With typical phosphors, most of the light emitted is phosphorescence, since the excitation and hence the fluorescence usually last just a fraction of a microsecond. A phosphor's *persistence* is defined as the time from the removal of excitation to the moment when phosphorescence has decayed to 10 percent of the initial light output. The range of persistence of different phosphors can reach many seconds, but for most phosphors used in graphics equipment it is usually 10 to 60 microseconds. This light output decays exponentially with time. Characteristics of phosphors are detailed in [SHER79].

The *refresh rate* of a CRT is the number of times per second the image is redrawn; it is typically 60 per second for raster displays. As the refresh rate decreases, *flicker* develops because the eye can no longer integrate the individual light impulses coming from a pixel. The refresh rate above which a picture stops flickering and fuses into a steady image is called the *critical fusion frequency*, or *CFF*. The process of fusion is familiar to all of us; it occurs whenever we watch television or motion pictures. A flicker-free picture appears constant or steady to the viewer, even though, in fact, any given point is "off" much longer than it is "on."

One determinant of the CFF is the phosphor's persistence: The longer the persistence, the lower the CFF. The relation between fusion frequency and persistence is nonlinear: Doubling persistence does not halve the CFF. As persistence increases into the several-second range, the fusion frequency becomes quite small. At the other extreme, even a phosphor with absolutely no persistence at all can be used, since all the eye really requires is to see some light for a short period of time, repeated at a frequency above the CFF.

Persistence is not the only factor affecting CFF. CFF also increases with image intensity and with ambient room lighting, and varies with different wavelengths of emitted light. Finally, it depends on the observer. Fusion is, after all, a physiological phenomenon, and differences among viewers of up to 20 Hz in CFF have been reported [ROGO83]. Cited fusion frequencies are thus usually averages for a large number of observers. Eliminating flicker for 99 percent of viewers of very high-intensity images (especially prevalent with black-on-white raster displays) requires refresh rates of 80 to 90 Hz.

The *horizontal scan rate* is the number of scan lines per second that the circuitry driving a CRT is able to display. The rate is approximately the product of the refresh rate and the number of scan lines. For a given scan rate, an increase in the refresh rate means a decrease in the number of scan lines.

The *resolution* of a monochromatic CRT is defined just as is resolution for hardcopy devices. Resolution is usually measured with a *shrinking raster*: A known number of equally spaced parallel lines that alternate between black and white are displayed, and the interline spacing is uniformly decreased until the lines just begin to merge together into a uniform

Samsung/Google, EX1027, p. 176

field of gray. This merging happens at about the point where the interline spacing is equal to the diameter at which the spot intensity is 60 percent of the intensity at the center of the spot. The resolution is the distance between the two outermost lines, divided by the number of lines in the raster. There is a clear dependence between spot size and achievable resolution: The larger the spot size, the lower the achievable resolution.

In the shrinking-raster process, the interline spacing is decreased not by modifying the contents of a raster bitmap, but by changing the gain (amount of amplification) of the vertical or horizontal deflection amplifiers, depending on whether the vertical or horizontal resolution is being measured. These amplifiers control how large an area on the screen is covered by the bitmap image. Thus, CRT resolution is (properly) not a function of the bitmap resolution, but may be either higher or lower than that of the bitmap.

Resolution is not a constant. As the number of electrons in the beam increases, resolution tends to decrease, because a bright line is wider than a dim line. This effect is a result of *bloom*, the tendency of a phosphor's excitation to spread somewhat beyond the area being bombarded, and also occurs because the spot size of an intense electron beam is bigger than that of a weak beam. Vertical resolution on a raster monitor is determined primarily by spot size; if the vertical resolution is n lines per inch, the spot size needs to be about $1/n$ inches. Horizontal resolution (in which the line-pairs are vertical) is determined by both spot size and the speed with which the electron beam can be turned on and off as it moves horizontally across the screen. This rate is related to the bandwidth of the display, as discussed in the next paragraph. Research on defining the resolution of a display precisely and on our ability to perceive images is ongoing. The *modulation transfer function*, used extensively in this research, relates a device's input signal to its output signal [SNYD85].

The bandwidth of a monitor has to do with the speed with which the electron gun can be turned on or off. To achieve a horizontal resolution of n pixels per scan line, it must be possible to turn the electron gun on at least $n/2$ times and off another $n/2$ times in one scan line, in order to create alternating on and off lines. Consider a raster scan of 1000 lines by 1000 pixels, displayed at a 60-Hz refresh rate. One pixel is drawn in about 11 nanoseconds [WHIT84], so the period of an on–off cycle is about 22 nanoseconds, which corresponds to a frequency of 45 MHz. This frequency is the minimum bandwidth needed to achieve 1000 lines (500 line-pairs) of resolution, but is not the actual bandwidth because we have ignored the effect of spot size. The nonzero spot size must be compensated for with a higher bandwidth which causes the beam to turn on and off more quickly, giving the pixels sharper edges than they would have otherwise. It is not unusual for the actual bandwidth of a 1000 by 1000 monitor to be 100 MHz. The actual relationships among resolution, bandwidth, and spot size are complex, and only recently has progress been made in quantifying them [INFA85].

Color television sets and color raster displays use some form of *shadow-mask CRT*. Here, the inside of the tube's viewing surface is covered with closely spaced groups of red, green, and blue phosphor dots. The dot groups are so small that light emanating from the individual dots is perceived by the viewer as a mixture of the three colors. Thus, a wide range of colors can be produced by each group, depending on how strongly each individual phosphor dot is excited. A *shadow mask*, which is a thin metal plate perforated with many small holes and mounted close to the viewing surface, is carefully aligned so that each of

Samsung/Google, EX1027, p. 177

the three electron beams (one each for red, green, and blue) can hit only one type of phosphor dot. The dots thus can be excited selectively.

Figure 4.14 shows one of the most common types of shadow-mask CRT, a *delta–delta* CRT. The phosphor dots are arranged in a triangular *triad* pattern, as are the three electron guns. The guns are deflected together, and are aimed (converged) at the same point on the viewing surface. The shadow mask has one small hole for each triad. The holes are precisely aligned with respect to both the triads and the electron guns, so that each dot in the triad is exposed to electrons from only one gun. High-precision delta–delta CRTs are particularly difficult to keep in alignment. An alternative arrangement, the *precision in-line delta CRT* shown in Fig. 4.15, is easier to converge and is gaining in popularity for high-precision (1000-scan-lines) monitors. In this case, the three beams simultaneously expose three in-line phosphor dots. However, the in-line arrangement does slightly reduce image sharpness at the edges of the tube. Still in the research laboratory but likely to become commercially viable is the *flat-panel color CRT*, in which the electron beams move parallel to the viewing surface, and are then turned 90° to strike the surface.

The need for the shadow mask and triads imposes a limit on the resolution of color CRTs not present with monochrome CRTs. In very high-resolution tubes, the triads are placed on about 0.21-millimeter centers; those in home television tubes are on about 0.60-millimeter centers (this distance is also called the *pitch* of the tube). Because a finely focused beam cannot be guaranteed to hit exactly in the center of a shadow-mask hole, the beam diameter (the diameter at which the intensity is 50 percent of the maximum) must be about $\frac{7}{4}$ times the pitch. Thus, on a mask with a pitch of 0.25-millimeter (0.01 inches),

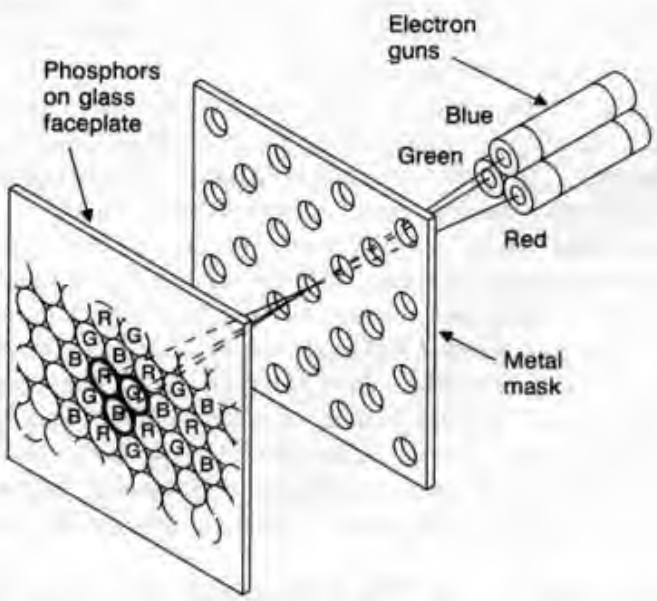


Fig. 4.14 Delta-delta shadow-mask CRT. The three guns and phosphor dots are arranged in a triangular (delta) pattern. The shadow mask allows electrons from each gun to hit only the corresponding phosphor dots.

Samsung/Google, EX1027, p. 178

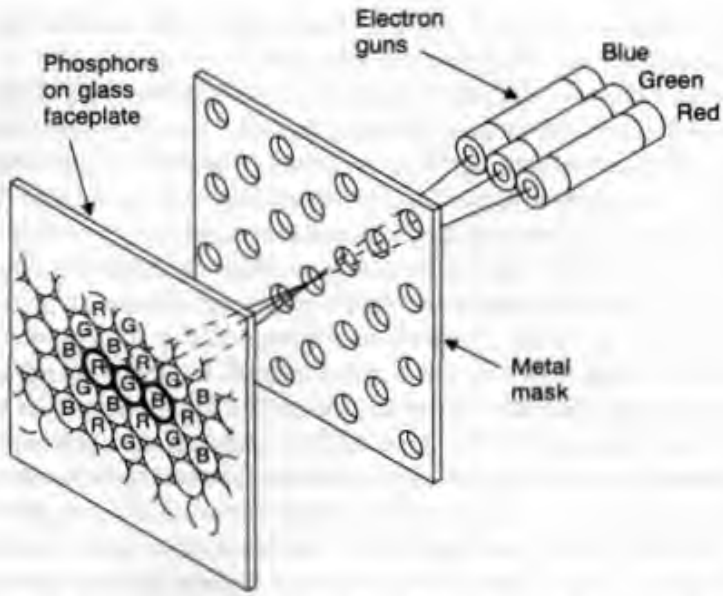


Fig. 4.15 A precision in-line CRT: the three electron guns are in a line.

the beam is about 0.018 inches across, and the resolution can be no more than about $\frac{1}{0.018} = 55$ lines per inch. On a 0.25-millimeter pitch, 19-inch (diagonal measure) monitor, which is about 15.5 inches wide by 11.6 inches high [CONR85], the resolution achievable is thus only $15.5 \times 55 = 850$ by $11.6 \times 55 = 638$. This value compares with a typical addressability of 1280 by 1024, or 1024 by 800. As illustrated in Fig. 4.2, a resolution somewhat less than the addressability is useful.

The pitch of the shadow mask is clearly an important limit on the resolution of shadow-mask CRTs. As pitch decreases, resolution can increase (assuming bandwidth and dot size are appropriate). The smaller the pitch is, however, the more difficult the tube is to manufacture. A small-pitch shadow mask is more fragile, making it more difficult to mount. It is also more likely to warp from heating by the electron beam. The *flat-tension-mask* tube has a flat faceplate, with the mask stretched tightly to maintain its position; a pitch of 0.15 millimeter is achievable with this technology.

The shadow mask also limits CRT brightness. Typically, only 20 percent of the electrons in the beam hit the phosphors—the rest hit the mask. Thus, fewer electrons make light than in a monochrome CRT. The number of electrons in the beam (the *beam current*) can be increased, but a higher current makes focusing more difficult and also generates more heat on the shadow mask, further exacerbating mask warping. Because the flat tension mask is more resistant to heating distortions, it allows a higher beam current and hence a brighter image.

Most high-quality shadow-mask CRTs have diagonals of 15 to 21 inches, with slightly curved faceplates that create optical distortions for the viewer. Several types of flat-face CRTs are becoming available, including a 29-inch–diagonal tube with a pitch of 0.31 millimeter. Of course, the price is high, but it will come down as demand develops.

Samsung/Google, EX1027, p. 179

The *direct-view storage tube* (DVST) is similar to the standard CRT, except that it does not need to be refreshed because the image is stored as a distribution of charges on the inside surface of the screen. Because no refresh is needed, the DVST can display complex images without the high scan rate and bandwidth required by a conventional CRT. The major disadvantage of the DVST is that modifying any part of an image requires redrawing the entire modified image to establish a new charge distribution in the DVST. This redraw can be unacceptably slow (many seconds for a complex image).

The ubiquitous Tektronix 4010 display terminal, based on the DVST, was the first low-cost, widely available interactive graphics terminal. It was the Model T of computer graphics, and was so pervasive that its instruction set became a defacto standard. Even today, many display systems include a Tektronix-compatibility feature, so that buyers can continue to run their (often large) libraries of older software developed for the 4010. Now, however, DVSTs have been superseded by raster displays and have essentially disappeared from the graphics scene.

A *liquid-crystal display* (LCD) is made up of six layers, as shown in Fig. 4.16. The front layer is a vertical polarizer plate. Next is a layer with thin grid wires electrodeposited on the surface adjoining the crystals. Next is a thin (about 0.0005-inch) liquid-crystal layer, then a layer with horizontal grid wires on the surface next to the crystals, then a horizontal polarizer, and finally a reflector.

The liquid-crystal material is made up of long crystalline molecules. The individual molecules normally are arranged in a spiral fashion such that the direction of polarization of polarized light passing through is rotated 90°. Light entering through the front layer is polarized vertically. As the light passes through the liquid crystal, the polarization is rotated 90° to horizontal, so the light now passes through the rear horizontal polarizer, is reflected, and returns through the two polarizers and crystal.

When the crystals are in an electric field, they all line up in the the same direction, and thus have no polarizing effect. Hence, crystals in the electric field do not change the polarization of the transmitted light, so the light remains vertically polarized and does not pass through the rear polarizer: The light is absorbed, so the viewer sees a dark spot on the display.

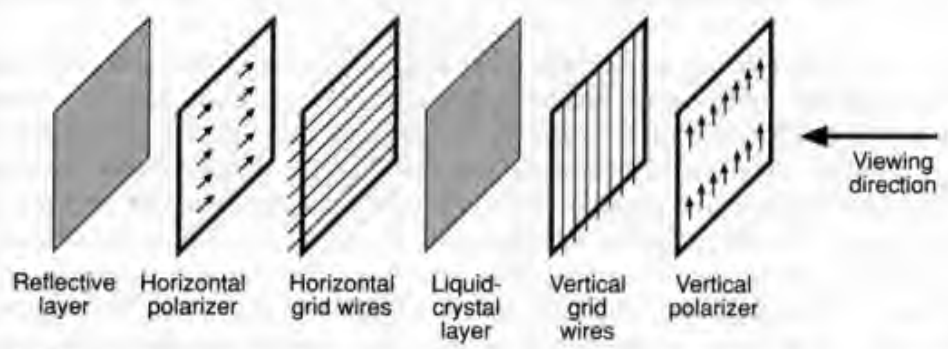


Fig. 4.16 The layers of a liquid-crystal display (LCD), all of which are sandwiched together to form a thin panel.

Samsung/Google, EX1027, p. 180

A dark spot at point (x_1, y_1) is created via *matrix addressing*. The point is selected by applying a negative voltage $-V$ to the horizontal grid wire x_1 and a positive voltage $+V$ to the vertical grid wire y_1: Neither $-V$ nor $+V$ is large enough to cause the crystals to line up, but their difference is large enough to do so. Now the crystals at (x_1, y_1) no longer rotate the direction of polarization of the transmitted light, so it remains vertically polarized and does not pass through the rear polarizer: The light is absorbed, so the viewer sees a dark spot on the display.

To display dots at (x_1, y_1) and (x_2, y_2), we cannot simply apply the positive voltage to x_1 and x_2 and the negative voltage to y_1 and y_2: that would cause dots to appear at (x_1, y_1), (x_1, y_2), (x_2, y_1), and (x_2, y_2), because all these points will be affected by the voltage. Rather, the points must be selected in succession, one after the other, and this selection process must be repeated, to refresh the activation of each point. Of course, if $y_1 = y_2$, then both points on the row can be selected at the same time.

The display is refreshed one row at a time, in raster-scan fashion. Those points in a row that are "on" (i.e., dark in the case of a black-on-white LCD display) are selected only about $1/N$ of the time, where N is the number of rows. Fortunately, once the crystals are lined up, they stay that way for several hundred milliseconds, even when the voltage is withdrawn (the crystals' equivalent of phosphors' persistence). But even so, the crystal is not switched on all the time.

Active matrix panels are LCD panels that have a transistor at each (x, y) grid point. The transistors are used to cause the crystals to change their state quickly, and also to control the degree to which the state has been changed. These two properties allow LCDs to be used in miniature television sets with continuous-tone images. The crystals can also be dyed to provide color. Most important, the transistor can serve as a memory for the state of a cell and can hold the cell in that state until it is changed. That is, the memory provided by the transistor enables a cell to remain on all the time and hence to be brighter than it would be if it had to be refreshed periodically. Color LCD panels with resolutions up to 800 by 1000 on a 14-inch diagonal panel have been built.

Advantages of LCDs are low cost, low weight, small size, and low power consumption. In the past, the major disadvantage was that LCDs were passive, reflecting only incident light and creating no light of their own (although this could be corrected with backlighting): Any glare washed out the image. In recent years, use of active panels has removed this concern.

Nonactive LCD technology has been adapted to color displays, and is sold commercially as the Tektronix liquid-crystal shutter (LCS). The LCS, placed in front of a standard black-and-white CRT, consists of three layers. The back layer, closest to the CRT, is a vertical polarizer, to polarize light emitted from the CRT. The layer also has a thin coating of a transparent conducting material. The next layer is the liquid crystal, and the third (front) layer is a color polarizer that transmits vertically polarized light as red and horizontally polarized light as green. This front layer also has a thin coating of the transparent conductor. If the crystals are in their normal state, they rotate the polarization plane by 90°, so the light is horizontally polarized as it approaches the color polarizer of the third layer, and is seen as green. If the appropriate voltage is applied to the conductive coatings on the front and back layers, then the crystals line up and do not affect the vertical polarization, so the light is seen as red.

Samsung/Google, EX1027, p. 181

The crystals are switched back and forth between their two states at a rate of 60 Hz. At the same time and in synchrony, images to be seen as red and green are alternated on the monochrome display. Mixtures of red and green are created by intensifying the same spot during the red and green phases, potentially with different intensities.

The LCS is an alternative to the shadow-mask CRT, but has limited color resolution. It is possible, however, that this technology can be extended to work with three colors. If it can be, the shadow mask will no longer be a limiting factor in achieving higher-resolution full-color displays. Spot size and bandwidth will be the major determinants, as with monochrome CRTs. Eliminating the shadow mask also will increase ruggedness. Because LCD displays are small and light, they can be used in head-mounted displays such as that discussed in Section 8.1.6.

The *plasma panel* is an array of tiny neon bulbs. Each bulb can be put into an "on" (intensified) state or an "off" state, and remains in the state until explicitly changed to the other. This memory property means that plasma panels need not be refreshed. Plasma panels typically have 50 to 125 cells per inch, with a 10- to 15-inch diagonal, but 40- by 40-inch panels with 50 cells per inch are sold commercially, and even larger and denser panels can be custom-made.

The neon bulbs are not discrete units, but rather are part of a single integrated panel made of three layers of glass, as seen in Fig. 4.17. The inside surface of the front layer has thin vertical strips of an electrical conductor; the center layer has a number of holes (the bulbs), and the inside surface of the rear layer has thin horizontal strips of an electrical conductor. Matrix addressing is used to turn bulbs on and off. To turn on a bulb, the system adjusts the voltages on the corresponding lines such that their difference is large enough to pull electrons from the neon molecules, thus firing the bulb and making it glow. Once the glow starts, a lower voltage is applied to sustain it. To turn off a bulb, the system momentarily decreases the voltages on the appropriate lines to less than the sustaining

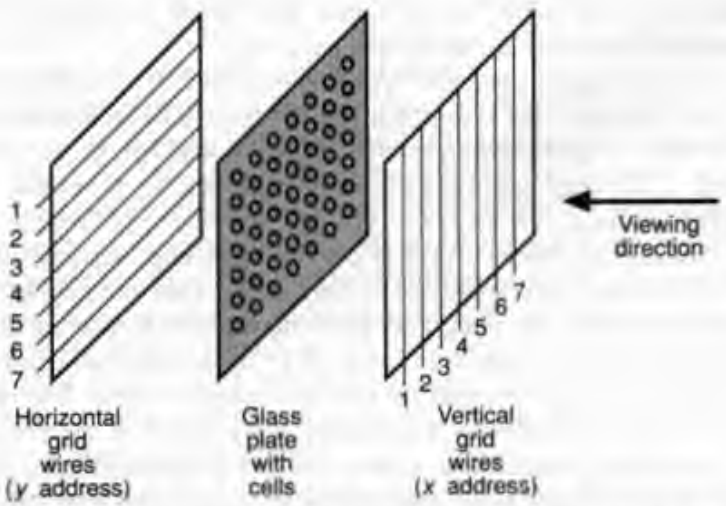


Fig. 4.17 The layers of a plasma display, all of which are sandwiched together to form a thin panel.

Samsung/Google, EX1027, p. 182

voltage. Bulbs can be turned on or off in about 15 microseconds. In some panel designs, the individual cells are replaced with an open cavity, because the neon glow is contained in a localized area. In this case, the front and back glass layers are separated by spacers. Some plasma panels can also display multiple gray levels.

The plasma panel has the advantages of being flat, transparent, and rugged, and of not needing a bitmap refresh buffer. It can be used with a rear-projection system to mix photographic slides as static background for computer-generated dynamic graphics, but has found most use in military applications, where small size and ruggedness are important. However, its cost, although continually decreasing, is still relatively high for its limited resolution. Color has been demonstrated in the laboratory, but is not commercially available.

Electroluminescent (EL) displays consist of the same gridlike structure as used in LCD and plasma displays. Between the front and back panels is a thin (typically 500-nanometers) layer of an electroluminescent material, such as zinc sulfide doped with manganese, that emits light when in a high electric field (about 10^6 volts per centimeter). A point on the panel is illuminated via the matrix-addressing scheme, several hundred volts being placed across the horizontal and vertical selection lines. Color electroluminescent displays are also available.

These displays are bright and can be switched on and off quickly, and transistors at each pixel can be used to store the image. Typical panel sizes are 6 by 8 inches up to 12 by 16 inches, with 70 addressable dots per inch. These displays' major disadvantage is that their power consumption is higher than that of the LCD panel. However, their brightness has led to their use in some portable computers.

Electrophoretic displays use positively charged colored particles suspended in a solution of a contrasting color, sealed between two parallel, closely spaced plates that have matrix-addressing selection lines. A negative voltage on the front selection line and a positive voltage on the rear selection line pulls the colored particles toward the front plate, where they are seen instead of the colored liquid. Reversing the voltages pulls the particles toward the rear plate, so the colored liquid is seen. The display has a memory: The particles stay where they have been placed until moved explicitly.

Most large-screen displays use some form of *projection CRT*, in which the light from a small (several-inch-diameter) but very bright monchrome CRT is magnified and projected from a curved mirror. Color systems use three projectors with red, green, and blue filters. A shadow-mask CRT does not create enough light to be projected onto a large (2-meter-diagonal) screen.

The *GE light-valve projection system* is used for very large screens, where the light output from the projection CRT would not be sufficient. A light valve is just what its name implies: A mechanism for controlling how much light passes through a valve. The light source can have much higher intensity than a CRT. In the most common approach, an electron gun traces an image on a thin oil film on a piece of glass. The electron charge causes the film to vary in thickness: A negatively charged area of the film is "stretched out," as the electrons repel one another, causing the film to become thinner. Light from the high-intensity source is directed at the glass, and is refracted in different directions because of the variation in the thickness of the oil film. Optics involving Schlieren bars and lenses project light that is refracted in certain directions on the screen, while other light is not

Samsung/Google, EX1027, p. 183

TABLE 4.3 COMPARISON OF DISPLAY TECHNOLOGIES

	CRT	Electro-luminescent	Liquid crystal	Plasma panel
power consumption	fair	fair–good	excellent	fair
screen size	excellent	good	fair	excellent
depth	poor	excellent	excellent	good
weight	poor	excellent	excellent	excellent
ruggedness	fair–good	good–excellent	excellent	excellent
brightness	excellent	excellent	fair–good	excellent
addressability	good–excellent	good	fair–good	good
contrast	good–excellent	good	fair	good
intensity levels per dot	excellent	fair	fair	fair
viewing angle	excellent	good	poor	good–excellent
color capability	excellent	good	good	fair
relative cost range	low	medium–high	low	high

projected. Color is possible with these systems, through use of either three projectors or a more sophisticated set of optics with a single projector. More details are given in [SHER79]. Other similar light-valve systems use LCDs to modulate the light beam.

Table 4.3 summarizes the characteristics of the four major display technologies. The pace of technological innovation is such, however, that some of the relationships may change over the next few years. Also, note that the liquid-crystal comparisons are for passive addressing; with active matrix addressing; gray levels and colors are achievable. More detailed information on these display technologies is given in [APT85; BALD85; CONR85; PERR85; SHER79; and TANN85].

4.3 RASTER-SCAN DISPLAY SYSTEMS

The basic concepts of raster graphics systems were presented in Chapter 1, and Chapter 2 provided further insight into the types of operations possible with a raster display. In this section, we discuss the various elements of a raster display, stressing two fundamental ways in which various raster systems differ one from another.

First, most raster displays have some specialized hardware to assist in scan converting output primitives into the pixmap, and to perform the raster operations of moving, copying, and modifying pixels or blocks of pixels. We call this hardware a *graphics display processor*. The fundamental difference among display systems is how much the display processor does versus how much must be done by the graphics subroutine package executing on the general-purpose CPU that drives the raster display. Note that the graphics display processor is also sometimes called a *graphics controller* (emphasizing its similarity to the control units for other peripheral devices) or a *display coprocessor*. The second key differentiator in raster systems is the relationship between the pixmap and the address space of the general-purpose computer's memory, whether the pixmap is part of the general-purpose computer's memory or is separate.

In Section 4.3.1, we introduce a simple raster display consisting of a CPU containing the pixmap as part of its memory, and a video controller driving a CRT. There is no display processor, so the CPU does both the application and graphics work. In Section 4.3.2, a

Samsung/Google, EX1027, p. 184

graphics processor with a separate pixmap is introduced, and a wide range of graphics-processor functionalities is discussed in Section 4.3.3. Section 4.3.4 discusses ways in which the pixmap can be integrated back into the CPU's address space, given the existence of a graphics processor.

4.3.1 Simple Raster Display System

The simplest and most common raster display system organization is shown in Fig. 4.18. The relation between memory and the CPU is exactly the same as in a nongraphics computer system. However, a portion of the memory also serves as the pixmap. The video controller displays the image defined in the frame buffer, accessing the memory through a separate access port as often as the raster-scan rate dictates. In many systems, a fixed portion of memory is permanently allocated to the frame buffer, whereas some systems have several interchangeable memory areas (sometimes called *pages* in the personal-computer world). Yet other systems can designate (via a register) any part of memory for the frame buffer. In this case, the system may be organized as shown in Fig. 4.19, or the entire system memory may be dual-ported.

The application program and graphics subroutine package share the system memory and are executed by the CPU. The graphics package includes scan-conversion procedures, so that when the application program calls, say, SRGP_lineCoord (*x1*, *y1*, *x2*, *y2*), the graphics package can set the appropriate pixels in the frame buffer (details on scan-conversion procedures were given in Chapter 3). Because the frame buffer is in the address space of the CPU, the graphics package can easily access it to set pixels and to implement the PixBlt instructions described in Chapter 2.

The video controller cycles through the frame buffer, one scan line at a time, typically 60 times per second. Memory reference addresses are generated in synchrony with the raster scan, and the contents of the memory are used to control the CRT beam's intensity or

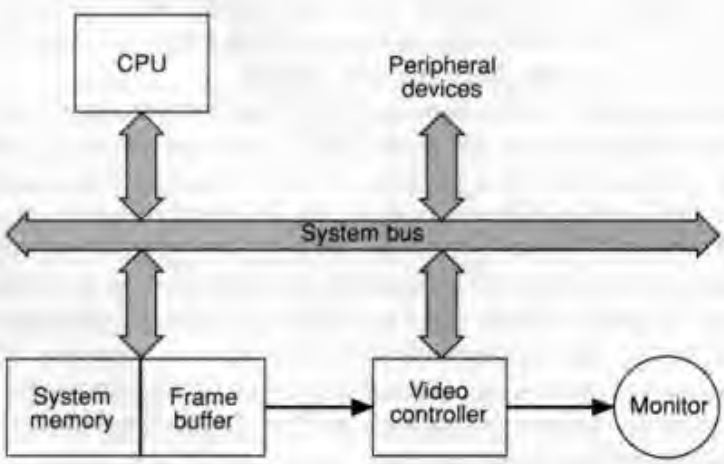


Fig. 4.18 A common raster display system architecture. A dedicated portion of the system memory is dual-ported, so that it can be accessed directly by the video controller, without the system bus being tied up.

Samsung/Google, EX1027, p. 185

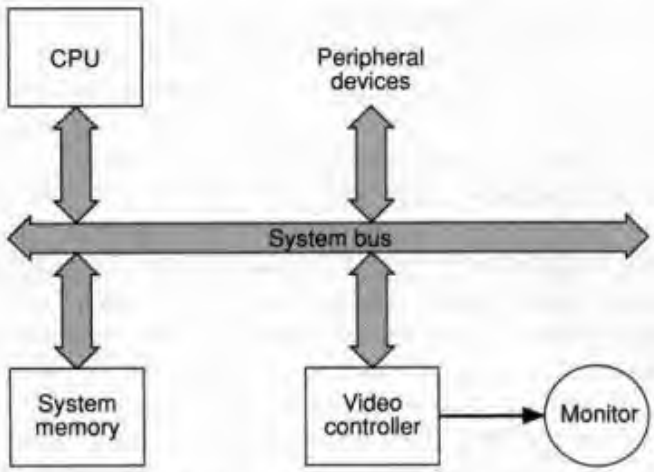


Fig. 4.19 A simple raster display system architecture. Because the frame buffer may be stored anywhere in system memory, the video controller accesses the memory via the system bus.

color. The video controller is organized as shown in Fig. 4.20. The raster-scan generator produces deflection signals that generate the raster scan; it also controls the X and Y address registers, which in turn define the memory location to be accessed next.

Assume that the frame buffer is addressed in x from 0 to x_{max} and in y from 0 to y_{max}; then, at the start of a refresh cycle, the X address register is set to zero and the Y register is set to y_{max} (the top scan line). As the first scan line is generated, the X address is incremented up through x_{max}. Each pixel value is fetched and is used to control the intensity of the CRT beam. After the first scan line, the X address is reset to zero and the Y address is decremented by one. The process continues until the last scan line ($y = 0$) is generated.

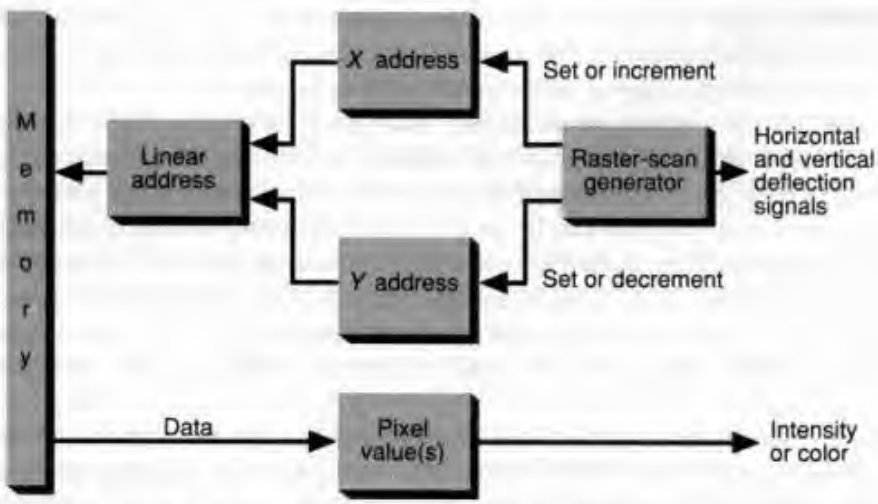


Fig. 4.20 Logical organization of the video controller.

Samsung/Google, EX1027, p. 186

In this simplistic situation, one memory access is made to the frame buffer for each pixel to be displayed. For a medium-resolution display of 640 pixels by 480 lines refreshed 60 times per second, a simple way to estimate the time available for displaying a single 1-bit pixel is to calculate $1/(480 \times 640 \times 60) = 54$ nanoseconds. This calculation ignores the fact that pixels are not being displayed during horizontal and vertical retrace (see Exercise 4.10). But typical RAM memory chips have cycle times around 200 nanoseconds: They cannot support one access each 54 nanoseconds! Thus, the video controller must fetch multiple pixel values in one memory cycle. In the case at hand, the controller might fetch 16 bits in one memory cycle, thereby taking care of 16 pixels $\times$ 54 ns/pixel = 864 nanoseconds of refresh time. The 16 bits are loaded into a register on the video controller, then are shifted out to control the CRT beam intensity, one each 54 nanoseconds. In the 864 nanoseconds this takes, there is time for about four memory cycles: one for the video controller and three for the CPU. This sharing may force the CPU to wait for memory accesses, potentially reducing the speed of the CPU by 25 percent. Of course, cache memory on the CPU chip can be used to ameliorate this problem.

It may not be possible to fetch 16 pixels in one memory cycle. Consider the situation when the pixmap is implemented with five 64-KB–memory chips, with each chip able to deliver 1 bit per cycle (this is called a 64-KB by 1 chip organization), for a total of 5 bits in the 200-nanoseconds cycle time. This is an average of 40 nanoseconds per bit (i.e., per pixel), which is not much faster than the 54 nanoseconds/pixel scan rate and leaves hardly any time for accesses to the memory by the CPU (except during the approximately 7-microsecond inter–scan-line retrace time and 1250-microsecond interframe vertical retrace time). With five 32-KB by 2 chips, however, 10 pixels are delivered in 200 nanoseconds, leaving slightly over half the time available for the CPU. With a 1600 by 1200 display, the pixel time is $1/(1600 \times 1200 \times 60) = 8.7$ nanoseconds. With a 200-nanoseconds memory cycle time, $200/8.7 = 23$ pixels must be fetched each cycle. A 1600 $\times$ 1200 display needs 1.92 MB of memory, which can be provided by eight 256-KB chips. Again, 256-KB by 1 chips can provide only 8 pixels per cycle: on the other hand, 32-KB by 8 chips can deliver 64 pixels, freeing two-thirds of the memory cycles for the CPU.

Access to memory by the CPU and video controller is clearly a problem: Table 4.4 shows that problem's magnitude. The solution is RAM architectures that accommodate the needs of raster displays. Chapter 18 discusses these architectures.

We have thus far assumed monochrome, 1-bit-per-pixel bitmaps. This assumption is fine for some applications, but is grossly unsatisfactory for others. Additional control over the intensity of each pixel is obtained by storing multiple bits for each pixel: 2 bits yield four intensities, and so on. The bits can be used to control not only intensity, but also color. How many bits per pixel are needed for a stored image to be perceived as having continuous shades of gray? Five or 6 bits are often enough, but 8 or more bits can be necessary. Thus, for color displays, a somewhat simplified argument suggests that three times as many bits are needed: 8 bits for each of the three additive primary colors red, blue, and green (see Chapter 13).

Systems with 24 bits per pixel are still relatively expensive, however, despite the decreasing cost of solid-state RAM. Furthermore, many color applications do not require 2^{24} different colors in a single picture (which typically has only 2^{18} to 2^{20} pixels). On the other hand, there is often need for both a small number of colors in a given picture or

Samsung/Google, EX1027, p. 187

TABLE 4.4 PERCENTAGE OF TIME AN IMAGE IS BEING TRACED DURING WHICH THE PROCESSOR CAN ACCESS THE MEMORY CONTAINING THE BITMAP*

Visible area pixels × lines	Chip size	Number of chips	Pixels per access	ns between accesses by video controller	Percent of time for processor accesses
512 × 512	256K × 1	1	1	64	0
512 × 512	128K × 2	1	2	127	0
512 × 512	64K × 4	1	4	254	20
512 × 512	32K × 8	1	8	507	60
512 × 512	16K × 16	1	16	1017	80
1024 × 1024	256K × 1	4	4	64	0
1024 × 1024	128K × 2	4	8	127	0
1024 × 1024	64K × 4	4	16	254	20
1024 × 1024	32K × 8	4	32	407	60
1024 × 1024	16K × 16	4	64	1017	80
1024 × 1024	1M × 1	1	1	16	0
1024 × 1024	64K × 16	1	16	254	21
1024 × 1024	32K × 32	1	32	509	61

*A 200-nanosecond memory cycle time and 60-Hz display rate are assumed throughout. The pixel time for a 512 × 512 display is assumed to be 64 nanoseconds; that for 1024 × 1024, 16 nanoseconds. These times are liberal, since they do not include the horizontal and vertical retrace times; the pixel times are actually about 45 and 11.5 nanoseconds, respectively.

application and the ability to change colors from picture to picture or from application to application. Also, in many image-analysis and image-enhancement applications, it is desirable to change the visual appearance of an image without changing the underlying data defining the image, in order, say, to display all pixels with values below some threshold as black, to expand an intensity range, or to create a pseudocolor display of a monochromatic image.

For these various reasons, the video controller of raster displays often includes a *video look-up table* (also called a *look-up table* or *LUT*). The look-up table has as many entries as there are pixel values. A pixel's value is used not to control the beam directly, but rather as an index into the look-up table. The table entry's value is used to control the intensity or color of the CRT. A pixel value of 67 would thus cause the contents of table entry 67 to be accessed and used to control the CRT beam. This look-up operation is done for each pixel on each display cycle, so the table must be accessible quickly, and the CPU must be able to load the look-up table on program command.

In Fig. 4.21, the look-up table is interposed between the frame buffer and the CRT display. The frame buffer has 8 bits per pixel, and the look-up table therefore has 256 entries.

The simple raster display system organizations of Figs. 4.18 and 4.19 are used in many inexpensive personal computers. Such a system is inexpensive to build, but has a number of disadvantages. First, scan conversion in software is slow. For instance, the (x, y) address of each pixel on a line must be calculated, then must be translated into a memory address consisting of a byte and bit-within-byte pair. Although each of the individual steps is simple, each is repeated many times. Software-based scan conversion slows down the

Samsung/Google, EX1027, p. 188

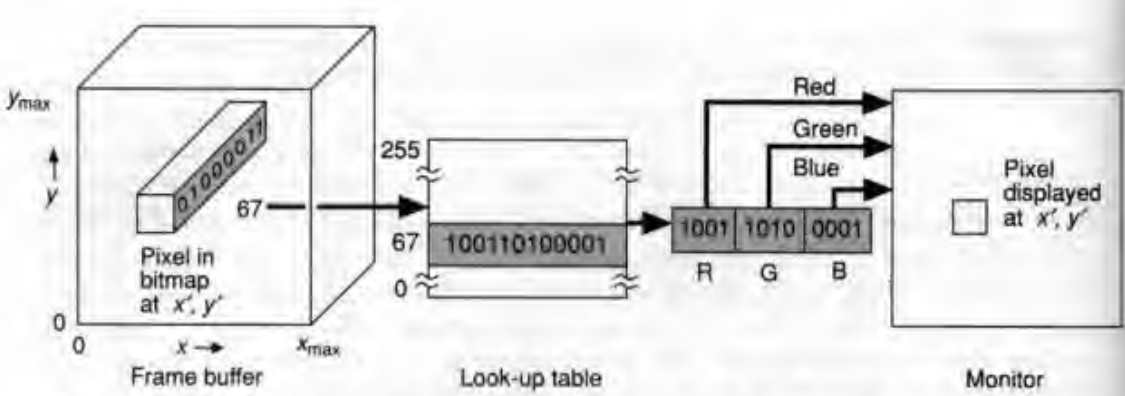


Fig. 4.21 Organization of a video look-up table. A pixel with value 67 (binary 01000011) is displayed on the screen with the red electron gun at $\frac{9}{15}$ of maximum, green at $\frac{10}{15}$, and blue at $\frac{1}{15}$. This look-up table is shown with 12 bits per entry. Up to 24 bits are common.

overall pace of user interaction with the application, potentially creating user dissatisfaction.

The second disadvantage of this architecture is that as the addressability or the refresh rate of the display increases, the number of memory accesses made by the video controller also increases, thus decreasing the number of memory cycles available to the CPU. The CPU is thus slowed down, especially with the architecture in Fig. 4.19. With the dual-porting of part of the system memory shown in Fig. 4.18, the slowdown occurs only when the CPU is accessing the frame buffer, usually for scan conversion or raster operations. These two disadvantages must be weighed against the ease with which the CPU can access the frame buffer and against the architectural simplicity of the system.

4.3.2 Raster Display System with Peripheral Display Processor

The raster display system with a peripheral display processor is a common architecture (see Fig. 4.22) that avoids the disadvantages of the simple raster display by introducing a separate graphics processor to perform graphics functions such as scan conversion and raster operations, and a separate frame buffer for image refresh. We now have two processors: the general-purpose CPU and the special-purpose display processor. We also have three memory areas: the system memory, the display-processor memory, and the frame buffer. The system memory holds data plus those programs that execute on the CPU: the application program, graphics package, and operating system. Similarly, the display-processor memory holds data plus the programs that perform scan conversion and raster operations. The frame buffer contains the displayable image created by the scan-conversion and raster operations.

In simple cases, the display processor can consist of specialized logic to perform the mapping from 2D (x, y) coordinates to a linear memory address. In this case, the scan-conversion and raster operations are still performed by the CPU, so the display-processor memory is not needed; only the frame buffer is present. Most peripheral display processors also perform scan conversion. In this section, we present a prototype system. Its

Samsung/Google, EX1027, p. 189

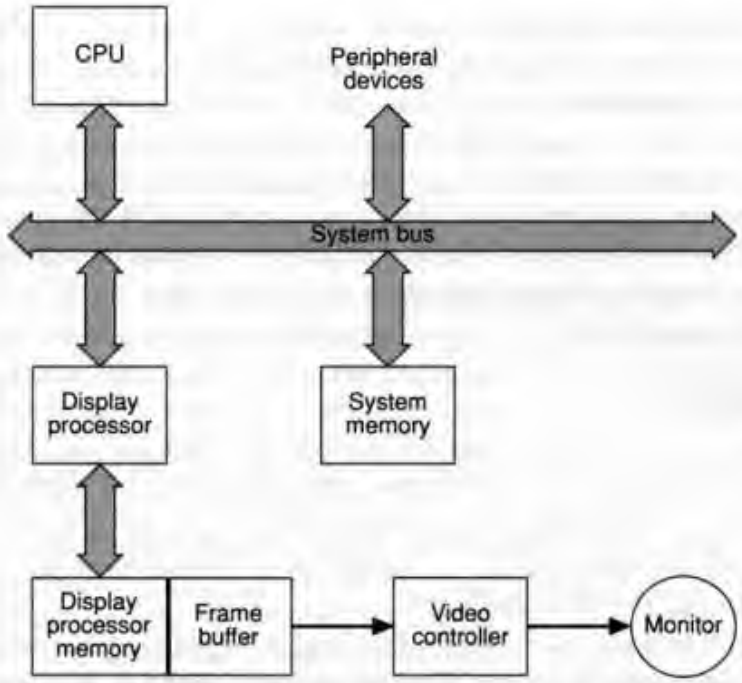


Fig. 4.22 Raster system architecture with a peripheral display processor.

features are a (sometimes simplified) composite of many typical commercially available systems, such as the plug-in graphics cards used with IBM's PC, XT, AT, PS, and compatible computers.

The frame buffer is 1024 by 1024 by 8 bits per pixel, and there is a 256-entry look-up table of 12 bits, 4 each for red, green, and blue. The origin is at lower left, but only the first 768 rows of the pixmap (y in the range of 0 to 767) are displayed. The display has six status registers, which are set by various instructions and affect the execution of other instructions. These are the CP (made up of the X and Y position registers), FILL, INDEX, WMODE, MASK, and PATTERN registers. Their operation is explained next.

The instructions for the simple raster display are as follows:

Move (x, y) The X and Y registers that define the current position (CP) are set to x and y. Because the pixmap is 1024 by 1024, x and y must be between 0 and 1023.

MoveR (dx, dy) The values dx and dy are added to the X and Y registers, thus defining a new CP. The dx and dy values must be between -1024 and $+1023$, and are represented in 2's-complement notation. The addition may cause overflow and hence a wraparound of the X and Y register values.

Line (x, y) A line is drawn from CP to (x, y), and this position becomes the new CP.

LineR (dx, dy) A line is drawn from CP to CP + (dx, dy), and this position becomes the new CP.

Point (x, y) The pixel at (x, y) is set, and this position becomes the new CP.

PointR (dx, dy) The pixel at CP + (dx, dy) is set, and this position becomes the new CP.

Samsung/Google, EX1027, p. 190

Rect (*x*, *y*) A rectangle is drawn between the CP and (*x*, *y*). The CP is unaffected.

RectR (*dx*, *dy*) A rectangle is drawn between the CP and CP + (*dx*, *dy*). The parameter *dx* can be thought of as the rectangle width, and *dy* as the height. The CP is unaffected.

Text (*n*, *address*) The *n* characters at memory location *address* are displayed, starting at the CP. Characters are defined on a 7- by 9-pixel grid, with 2 extra pixels of vertical and horizontal spacing to separate characters and lines. The CP is updated to the lower-left corner of the area in which character *n* + 1 would be displayed.

Circle (*radius*) A circle is drawn centered at the CP. The CP is unaffected.

Arc (*radius*, *startAngle*, *endAngle*) An arc of a circle centered at the CP is drawn. Angles are in tenths of a degree, measured positively counterclockwise from the *x* axis. The CP is unaffected.

CircleSector (*radius*, *startAngle*, *endAngle*) A "pie slice" is drawn consisting of a closed area with lines going from the CP to the endpoints of the arc. The CP is unaffected.

Polygon (*n*, *address*) At *address* is stored a vertex list $(x_1, y_1, x_2, y_2, x_3, y_3, \ldots, x_n, y_n)$. A polygon is drawn starting at (x_1, y_1), through all the vertexes up to (x_n, y_n), and then back to (x_1, y_1). The CP is unaffected.

AreaFill (*flag*) The *flag* is used to set the FILL flag in the raster display. When the flag is set to ON (by a nonzero value of *flag*), all the areas created by the commands Rect, RectR, Circle, CircleSector, Polygon are filled in as they are created, using the pattern defined with the Pattern command.

PixelValue (*index*) The pixel value *index* is loaded into the INDEX register. Its value is loaded into the pixmap when any of the output primitives in the preceding list are scan-converted.

Pattern (*row1*, *row2*, ..., *row16*) The 16 2-byte arguments define the pattern used to fill areas created by Rect, RectR, Circle, CircleSector, and Polygon. The pattern is a 16 by 16 array of bits, stored in the PATTERN register. When a filled area is being created and the FILL register is ON, then, if a bit in the PATTERN register is a 1, the pixel value in the INDEX register is loaded into the pixmap; otherwise, the pixmap is unaffected. A solid fill is created if all the bits of the PATTERN register are ON.

WBlockR (*dx*, *dy*, *address*) The 8-bit pixel values stored starting at *address* in main memory are written into the rectangular region of the pixmap from CP to CP + (*dx*, *dy*). Writing begins at the upper-left corner of the region and proceeds row by row, top to bottom.

RBlockR (*dx*, *dy*, *address*) A rectangular region of the pixmap, from CP to CP + (*dx*, *dy*), is read into the main memory area beginning at *address*. Reading begins at the upper-left corner of the region and proceeds row by row, top to bottom.

RasterOp (*dx*, *dy*, *xdest*, *ydest*) A rectangular region of the frame buffer, from CP to CP + (*dx*, *dy*), is combined with the region of the same size with lower-left corner at (*xdest*, *ydest*), overwriting that region. The combination is controlled by the WMODE register.

WMode (*mode*) The value of *mode* is loaded into the WMODE register. This register controls how pixels in the frame buffer are combined with any pixel value being written into the frame buffer. There are four *mode* values: **replace**, **xor**, **and**, and **or**. The

Samsung/Google, EX1027, p. 191

modes behave as described in Chapter 2. Note that, for simplicity, the preceding command descriptions have been written as though **replace** were the only write-mode value. In **xor** mode, new values written into the frame buffer are combined with the current value using a bit-by-bit exclusive-or operation.

Mask (*mask*) The 8-bit *mask* value is loaded into the MASK register. This register controls which bits of the frame buffer are modified during writing to the frame buffer: 1 allows modification of the corresponding bit; 0 inhibits modification.

LuT (*index, red, green, blue*) Entry *index* in the look-up table is loaded with the given color specification. The low-order 4 bits of each color component are loaded into the table.

Table 4.5 summarizes these commands. Notice that the MASK and WMODE registers affect *all* commands that write into the frame buffer.

The commands and immediate data are transferred to the display processor via a first-in, first-out (FIFO) buffer (i.e., a queue) in a dedicated portion of the CPU address space. The graphics package places commands into the queue, and the display accesses the instructions and executes them. Pointers to the start and end of the buffer are also in specific memory locations, accessible to both the CPU and display. The pointer to the start of the buffer is modified by the display processor each time a byte is removed; the pointer to the end of the buffer is modified by the CPU each time a byte is added. Appropriate testing is done to ensure that an empty buffer is not read and that a full buffer is not written. Direct memory access is used to fetch the addressed data for the instructions.

A queue is more attractive for command passing than is a single instruction register or location accessed by the display. First, the variable length of the instructions favors the queue concept. Second, the CPU can get ahead of the display, and queue up a number of display commands. When the CPU has finished issuing display commands, it can proceed to do other work while the display empties out the queue.

Programming examples. Programming the display is similar to using the SRGP package of Chapter 2, so only a few examples will be given here. A "Z" means that a hexadecimal value is being specified; an "A" means that the address of the following parenthetical list is used.

A white line on a completely black background is created as follows:

```
LuT          5, 0, 0, 0              Look-up-table entry 5 is black
LuT          6, Z'F', Z'F', Z'F'     Look-up-table entry 6 is white
WMode        replace
AreaFill     true                    Turn on the FILL flag
Pattern      32Z'FF'                 32 bytes of all 1s, for solid pattern
Mask         Z'FF'                   Enable writing to all planes
PixelValue   5                       Scan convert using pixel value of 5
Move         0, 0
Rect         1023, 767               Visible part of frame buffer now black
PixelValue   6                       Scan convert using pixel value of 6
Move         100, 100
LineR        500, 400                Draw the line
```

Samsung/Google, EX1027, p. 192

TABLE 4.5 SUMMARY OF THE RASTER DISPLAY COMMANDS*

Command mnemonic	Arguments and lengths	How CP is affected	Registers that affect command
Move	*x*(2), *y*(2)	CP := (*x*, *y*)	–
MoveR	*dx*(2), *dy*(2)	CP := CP + (*dx*, *dy*)	CP
Line	*x*(2), *y*(2)	CP := (*x*, *y*)	CP, INDEX, WMODE, MASK
LineR	*dx*(2), *dy*(2)	CP := CP + (*dx*, *dy*)	CP, INDEX, WMODE, MASK
Point	*x*(2), *y*(2)	CP := (*x*, *y*)	CP, INDEX, WMODE, MASK
PointR	*dx*(2), *dy*(2)	CP := CP + (*dx*, *dy*)	CP, INDEX, WMODE, MASK
Rect	*x*(2), *y*(2)	–	CP, INDEX, WMODE, MASK, FILL, PATTERN
RectR	*dx*(2), *dy*(2)	–	CP, INDEX, WMODE, MASK, FILL, PATTERN
Text	*n*(1), *address*(4)	CP := next char pos'n	CP, INDEX, WMODE, MASK
Circle	*radius*(2)	–	CP, INDEX, WMODE, MASK, FILL, PATTERN
Arc	*radius*(2), *startAngle*(2), *endAngle*(2)	–	CP, INDEX, WMODE, MASK
CircleSector	*radius*(2), *startAngle*(2), *endAngle*(2)	–	CP, INDEX, WMODE, MASK, FILL, PATTERN
Polygon	*n*(1), *address*(4)	–	CP, INDEX, WMODE, MASK, FILL, PATTERN
AreaFill	*flag*(1)	–	–
PixelValue	*index*(1)	–	–
Pattern	*address*(4)	–	–
WBlockR	*dx*(2), *dy*(2), *address*(4)	–	CP, INDEX, WMODE, MASK
RBlockR	*dx*(2), *dy*(2), *address*(4)	–	CP, INDEX, WMODE, MASK
RasterOp	*dx*(2), *dy*(2), *xdest*(2), *dest*(2)	–	CP, INDEX, WMODE, MASK
Mask	*mask*(1)	–	–
WMode	*mode*(1)	–	–
LuT	*index*(1), *red*(1), *green*(1), *blue*(1)	–	–

*The number in parentheses after each argument is the latter's length in bytes. Also indicated are the effect of the command on the CP, and which registers affect how the command operates.

Samsung/Google, EX1027, p. 193

A red circle overlapping a blue triangle, all on a black background, are created with:

LuT	5, 0, 0, 0	Look-up–table entry 5 is black
LuT	7, Z'F', 0, 0	Look-up–table entry 7 is red
LuT	8, 0, 0, Z'F'	Look-up–table entry 8 is blue
WMode	replace	
AreaFill	Z'FF'	On
Pattern	32Z'FF'	32 bytes of all 1s, for solid pattern
Mask	Z'FF'	Enable writing to all planes
PixelValue	5	Get ready for black rectangle
Move	0, 0	
Rect	1023, 767	Visible part of frame buffer now black
PixelValue	8	Next do blue triangle as a three-vertex polygon
Polygon	3, A(200, 200, 800, 200, 500, 700)	
PixelValue	7	Put red circle on top of triangle
Move	511, 383	Put CP at center of display
Circle	100	Radius-100 circle at CP

4.3.3 Additional Display-Processor Functionality

Our simple display processor performs only some of the graphics-related operations that might be implemented. The temptation faced by the system designer is to offload the main CPU more and more by adding functionality to the display processor, such as by using a local memory to store lists of display instructions, by doing clipping and the window-to-viewport transformation, and perhaps by providing pick-correlation logic and automatic feedback when a segment is picked. Ultimately, the display processor becomes another general-purpose CPU doing general interactive graphics work, and the designer is again tempted to provide special-function hardware to offload the display processor.

This *wheel of reincarnation* was identified by Myer and Sutherland in 1968 [MYER68]. These authors' point was that there is a tradeoff between special-purpose and general-purpose functionality. Special-purpose hardware usually does the job faster than does a general-purpose processor. On the other hand, special-purpose hardware is more expensive and cannot be used for other purposes. This tradeoff is an enduring theme in graphics system design.

If clipping (Chapter 3) is added to the display processor, then output primitives can be specified to the processor in coordinates other than device coordinates. This specification can be done in floating-point coordinates, although some display processors operate on only integers (this is changing rapidly as inexpensive floating-point chips become available). If only integers are used, the coordinates used by the application program must be integer, or the graphics package must map floating-point coordinates into integer coordinates. For this mapping to be possible, the application program must give the graphics package a rectangle guaranteed to enclose the coordinates of all output primitives specified to the package. The rectangle must then be mapped into the maximum integer range, so that everything within the rectangle is in the integer coordinate range.

If the subroutine package is 3D, then the display processor can perform the far more complex 3D geometric transformations and clipping described in Chapters 5 and 6. Also, if the package includes 3D surface primitives, such as polygonal areas, the display processor

Samsung/Google, EX1027, p. 194

can also perform the visible surface-determination and rendering steps discussed in Chapters 15 and 16. Chapter 18 discusses some of the fundamental approaches to organizing general- and special-purpose VLSI chips to perform these steps quickly. Many commercially available displays provide these features.

Another function that is often added to the display processor is *local segment storage*, also called *display list storage*. Display instructions, grouped into named segments and having unclipped integer coordinates, are stored in the display-processor memory, permitting the display processor to operate more autonomously from the CPU.

What exactly can a display processor do with these stored segments? It can transform and redraw them, as in zooming or scrolling. Local dragging of segments into new positions can be provided. Local picking can be implemented by having the display processor compare the cursor position to all the graphics primitives (more efficient ways of doing this are discussed in Chapter 7). Regeneration, required to fill in the holes created when a segment is erased, can also be done from segment storage. Segments can be created, deleted, edited, and made visible or invisible.

Segments can also be copied or referenced, both reducing the amount of information that must be sent from the CPU to the display processor and economizing on storage in the display processor itself. For instance, the display instructions to create a VLSI chip pad configuration to be used many times in a drawing can be sent to the display processor just once and can be stored as a segment. Each occurrence of the pad is then sent as a display instruction referencing the segment. It is possible to build up a complex hierarchical data structure using this capability, and many commercial display processors with local segment memory can copy or reference other segments. When the segments are displayed, a reference to another segment must be preceded by saving the display processor's current state, just as a subroutine call is preceded by saving the CPU's current state. References can be nested, giving rise to a *structured display file* or *hierarchical display list*, as in PHIGS [ANSI88], which is discussed further in Chapter 7. In the GKS graphics package [ENDE87; HOPG86], which uses a linear, unnested display list, an existing segment can be copied into a segment that is being created.

The segment data structures need not be in the display-processor memory (Fig. 4.22): They can be built directly by the graphics package in system memory and accessed by the display processor. This option, of course, requires that the display processor be able to read from system memory, which implies that the display processor must be directly on the system bus—RS-232 and Ethernet-speed connections are not viable.

If all the information being displayed is represented in a segment data structure, then the display processor can also implement window-manager operations such as move, open, close, resize, scroll, push, and pop. When a window is panned, the segments are retraversed with a new viewing transformation in effect. At some point, the wheel of reincarnation will again come into play, but we must also keep in mind the startlingly low cost of special-purpose VLSI chips: By the time this book becomes outdated, window manager chips are likely to cost only a few dollars each. Indeed, the pace of technological innovation is such that the graphics functionality found in display processors will continue its dramatic increase in scope and decrease in cost.

Although this raster display system architecture with its graphics display and separate frame buffer has many advantages over the simple raster display system of Section 4.3.1, it

Samsung/Google, EX1027, p. 195

also has some disadvantages. If the display processor is accessed by the CPU as a peripheral on a direct-memory-access port or on an RS-232 interface, then there is considerable operating-system overhead each time an instruction is passed to it (this is not an issue for a display processor whose instruction register is memory-mapped into the CPU's address space, since then it is easy for the graphics package to set up the registers directly).

There is also a marked partitioning of the memories, as was shown in Fig. 4.22. Building the display list in the display-list memory is slow because of the need to issue a display-processor instruction to add or delete elements. The display list may have to be duplicated in the main processor's memory, because it cannot always be read back. An environment in which the display list is built directly in main memory by the graphics subroutine package can thus be more flexible, faster, and easier to program.

The raster-operation command is a particular difficulty. Conceptually, it should have four potential source–destination pairs: system memory to system memory, system memory to frame buffer, frame buffer to system memory, and frame buffer to frame buffer (here, the frame buffer and display processor memory of Fig. 4.22 are considered identical, since they are in the same address space). In display-processor systems, however, the different source–destination pairs are handled in different ways, and the system-memory-to-system-memory case may not exist. This lack of symmetry complicates the programmer's task and reduces flexibility. For example, if the offscreen portion of the pixmap becomes filled with menus, fonts, and so on, then it is difficult to use main memory as an overflow area. Furthermore, because the use of pixmaps is so pervasive, failure to support raster operations on pixmaps stored in main memory is not really viable.

Another problem is that the output of scan-conversion algorithms *must* go to the frame buffer. This requirement precludes *double-buffering*: scan converting a new image into system memory, then copying it into the pixmap to replace the image currently stored there. In addition, certain window-manager strategies and animation techniques require partially or completely obscured windows to be kept current in offscreen canvases, again requiring scan conversion into system memory (Chapters 10 and 19).

The display processor defined earlier in this section, like many real display processors, moves raster images between the system memory and frame buffer via I/O transfers on the system bus. Unfortunately, this movement can be too slow for real-time operations, such as animation, dragging, and popping up windows and menus: The time taken in the operating system to initiate the transfers and the transfer rate on the bus get in the way. This problem can be partially relieved by increasing the display processor's memory to hold more offscreen pixmaps, but then that memory is not available for other purposes—and there is almost never enough memory anyway!

4.3.4 Raster Display System with Integrated Display Processor

We can ameliorate many of the shortcomings of the peripheral display processor discussed in the previous section by making the frame buffer part of the system memory, thus creating the *single-address-space* (SAS) display system architecture shown in Fig. 4.23. Here the display processor, the CPU, and the video controller are all on the system bus and can thus all access system memory. The origin and, in some cases, the size of the frame buffer are held in registers, making double-buffering a simple matter of reloading the register: The

Samsung/Google, EX1027, p. 196

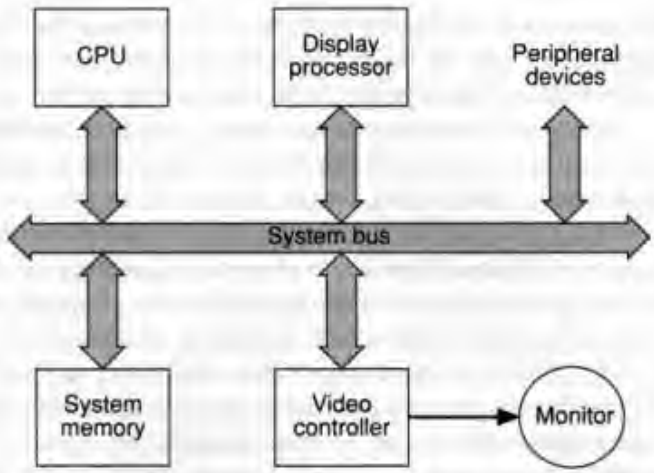


Fig. 4.23 A single-address-space (SAS) raster display system architecture with an integral display processor. The display processor may have a private memory for algorithms and working storage.

results of scan conversion can go either into the frame buffer for immediate display, or elsewhere in system memory for later display. Similarly, the source and destination for raster operations performed by the display processor can be anywhere in system memory (now the only memory of interest to us). This arrangement is also attractive because the CPU can directly manipulate pixels in the frame buffer simply by reading or writing the appropriate bits.

SAS architecture has, however, a number of shortcomings. Contention for access to the system memory is the most serious. We can solve this problem at least partially by dedicating a special portion of system memory to be the frame buffer and by providing a second access port to the frame buffer from the video controller, as shown in Fig. 4.24. Another solution is to use a CPU chip containing instruction- or data-cache memories, thus reducing the CPU's dependence on frequent and rapid access to the system memory. Of course, these and other solutions can be integrated in various ingenious ways, as discussed in more detail in Chapter 18. In the limit, the hardware PixBlt may work on only the frame buffer. What the application programmer sees as a single PixBlt instruction may be treated as several different cases, with software simulation if the source and destination are not supported by the hardware. Some processors are actually fast enough to do this, especially if they have an instruction-cache memory in which the tight inner loop of the software simulation can remain.

As suggested earlier, nontraditional memory-chip organizations for frame buffers also can help to avoid the memory-contention problem. One approach is to turn on all the pixels on a scan line in one access time, thus reducing the number of memory cycles needed to scan convert into memory, especially for filled areas. The video RAM (VRAM) organization, developed by Texas Instruments, can read out all the pixels on a scan line in one cycle, thus reducing the number of memory cycles needed to refresh the display. Again, Chapter 18 gives more detail on these issues.

Samsung/Google, EX1027, p. 197

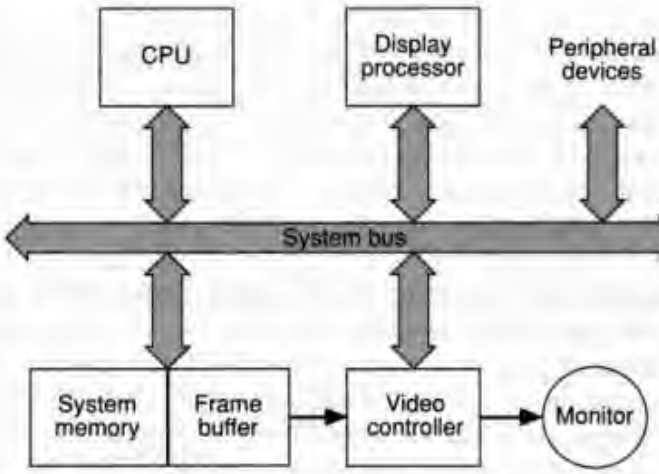


Fig. 4.24 A more common single-address-space raster display system architecture with an integral display processor (compare to Fig. 4.23). The display processor may have a private memory for algorithms and working storage. A dedicated portion of the system memory is dual-ported so that it can be accessed directly by the video controller, without the system bus being tied up.

Another design complication arises if the CPU has a virtual address space, as do the commonly used Motorola 680x0 and Intel 80x86 families, and various reduced-instruction-set-computer (RISC) processors. In this case memory addresses generated by the display processor must go through the same dynamic address translation as other memory addresses. In addition, many CPU architectures distinguish between a kernel operating system virtual address space and an application program virtual address space. It is often desirable for the frame buffer (canvas 0 in SRGP terminology) to be in the kernel space, so that the operating system's display device driver can access it directly. However, the canvases allocated by the application program must be in the application space. Therefore display instructions which access the frame buffer must distinguish between the kernel and application address spaces. If the kernel is to be accessed, then the display instruction must be invoked by a time-consuming operating system service call rather than by a simple subroutine call.

Despite these potential complications, more and more raster display systems do in fact have a single-address-space architecture, typically of the type in Fig. 4.24. The flexibility of allowing both the CPU and display processor to access any part of memory in a uniform and homogeneous way is very compelling, and does simplify programming.

4.4 THE VIDEO CONTROLLER

The most important task for the video controller is the constant refresh of the display. There are two fundamental types of refresh: *interlaced* and *noninterlaced*. The former is used in broadcast television and in raster displays designed to drive regular televisions. The refresh cycle is broken into two fields, each lasting $\frac{1}{60}$ second; thus, a full refresh lasts $\frac{1}{30}$ second. All

Samsung/Google, EX1027, p. 198

odd-numbered scan lines are displayed in the first field, and all even-numbered ones are displayed in the second. The purpose of the interlaced scan is to place some new information in all areas of the screen at a 60-Hz rate, since a 30-Hz refresh rate tends to cause flicker. The net effect of interlacing is to produce a picture whose effective refresh rate is closer to 60 than to 30 Hz. This technique works as long as adjacent scan lines do in fact display similar information; an image consisting of horizontal lines on alternating scan lines would flicker badly. Most video controllers refresh at 60 or more Hz and use a noninterlaced scan.

The output from the video controller has one of three forms: RGB, monochrome, or NTSC. For RGB (red, green, blue), separate cables carry the red, green, and blue signals to control the three electron guns of a shadow-mask CRT, and another cable carries the synchronization to signal the start of vertical and horizontal retrace. There are standards for the voltages, wave shapes, and synchronization timings of RGB signals. For 480–scan-line monochrome signals, RS-170 is the standard; for color, RS-170A; for higher-resolution monochrome signals, RS-343. Frequently, the synchronization timings are included on the same cable as the green signal, in which case the signals are called *composite video*. Monochrome signals use the same standards but have only intensity and synchronization cables, or merely a single cable carrying composite intensity and synchronization.

NTSC (National Television System Committee) video is the signal format used in North American commercial television. Color, intensity, and synchronization information is combined into a signal with a bandwidth of about 5 MHz, broadcast as 525 scan lines, in two fields of 262.5 lines each. Just 480 lines are visible; the rest occur during the vertical retrace periods at the end of each field. A monochrome television set uses the intensity and synchronization information; a color television set also uses the color information to control the three color guns. The bandwidth limit allows many different television channels to broadcast over the frequency range allocated to television. Unfortunately, this bandwidth limits picture quality to an effective resolution of about 350 by 350. Nevertheless, NTSC is the standard for videotape-recording equipment. Matters may improve, however, with increasing interest in 1000-line high-definition television (HDTV) for videotaping and satellite broadcasting. European and Soviet television broadcast and videotape standards are two 625–scan-line, 50-Hz standards, SECAM and PAL.

Some video controllers superimpose a programmable cursor, stored in a 16 by 16 or 32 by 32 pixmap, on top of the frame buffer. This avoids the need to PixBlt the cursor shape into the frame buffer each refresh cycle, slightly reducing CPU overhead. Similarly, some video controllers superimpose multiple small, fixed-size pixmaps (called *sprites*) on top of the frame buffer. This feature is used often in video games.

4.4.1 Animation with the Lookup Table

Raster images can be animated in several ways. To show a rotating object, we can scan convert into the pixmap successive views of the object from slightly different locations, one after the other. This scan conversion must be done at least 10 (preferably 15 to 20) times per second to give a reasonably smooth effect, and hence a new image must be created in no more than 100 milliseconds. However, if scan-converting the object takes much of this 100 milliseconds—say 75 milliseconds—then the complete object can be displayed for only 25

Samsung/Google, EX1027, p. 199

milliseconds before it must be erased and redrawn—a distracting effect. Double-buffering is used to avoid this problem. The frame buffer is divided into two images, each with half of the bits per pixel of the overall frame buffer. If we call the two halves of the pixmap *image0* and *image1*, we can describe the animation as follows:

```
Load look-up table to display all pixel value as background color;
Scan convert object into image0;
Load look-up table to display only image0;
do {
    Scan convert object into image1;
    Load look-up table to display only image1;
    Rotate object data structure description;
    Scan convert object into image0;
    Load look-up table to display only image0;
    Rotate object data structure description;
} while (not terminated);
```

Of course, if rotating and scan converting the object takes more than 100 milliseconds, the animation is quite jerky, but the transition from one image to the next appears to be instantaneous, as loading the look-up table typically takes less than 1 millisecond.

Another form of look-up–table animation displays a short, repetitive sequence of images [SHOU79]. Suppose we want to display a bouncing ball. Figure 4.25 shows how the frame buffer is loaded; the numbers indicate the pixel values placed in each region of the frame buffer. Figure 4.26 shows how to load the look-up table at each step to display all but one of the balls at background color 0. By cycling the contents of the look-up table, we can achieve motion effects. Not only can balls be bounced, but the effect of moving lights on movie marquees can be simulated, as can the motion of fluids through pipes and the rotation of wheels. This is discussed further in Section 21.1.4.

For more complicated cyclic animation, such as rotating a complex wire-frame object, it may be impossible to keep the separate images from overlapping in the frame buffer. In that case, some of the displayed images will have "holes" in them. A few such holes are not especially distracting, especially if realism is not essential. As more holes appear, however, the animation loses its effectiveness and double-buffering becomes more attractive.

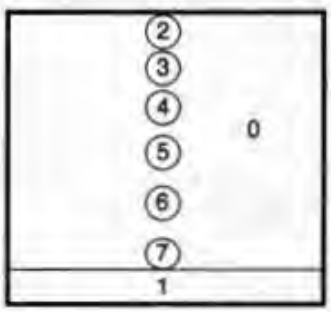


Fig. 4.25 Contents of the frame buffer for a bouncing-ball animation.

Samsung/Google, EX1027, p. 200

Entry number	Colors loaded in table at each step in animation										
	1	2	3	4	5	6	7	8	9	10	11
0	white	white	white	white	white	white	white	white	white	white	white
1	black	black	black	black	black	black	black	black	black	black	black
2	red										red
3		red								red	
4			red						red		
5				red				red			
6					red		red				
7						red					

Fig. 4.26 Look-up table to bounce a red ball on a black surface against a white background. Unspecified entries are all white.

4.4.2 Bitmap Transformations and Windowing

With some video controllers, the pixmap is decoupled from the view surface; that is, the direct, fixed correspondence between positions in the frame buffer and positions on the display is removed. An *image transformation* then defines the correspondence. The image transformation transforms from the frame buffer to the view surface. The transformation can, in general, include a translation, scaling, rotation, and an additional clipping operation.

Figure 4.27 shows the type of transformation found on some raster displays. Part of the frame buffer, defined by a rectangular clip region, is enlarged to fill the entire view surface. The ratio between the size of the window and the view surface must be an integer (3, in the figure). Pixels in the frame buffer outside the window are not used, and none of the pixel values are modified; the transformations are applied by the video controller at the refresh rate.

The image transformation can be changed many times per second so as to give the real-time–dynamics effect of scrolling over or zooming into an image. Also, a sequence of arbitrary images can be shown rapidly by loading each image into a different area of the frame buffer. The image transformation is changed periodically to display a full-screen view first of one area, then of the next, and so on.

The scaling needed to enlarge an image is trivially accomplished by repeating pixel values from within the window as the image is displayed. For a scale factor of 2, each pixel

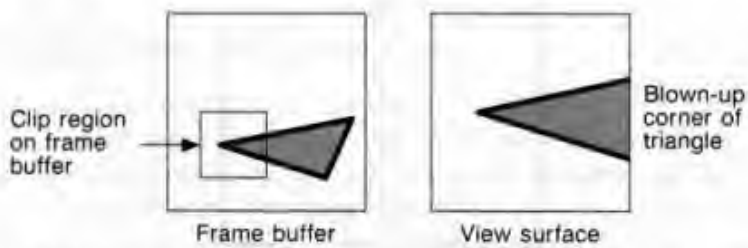


Fig. 4.27 Portion of the frame buffer enlarged on the view surface.

Samsung/Google, EX1027, p. 201

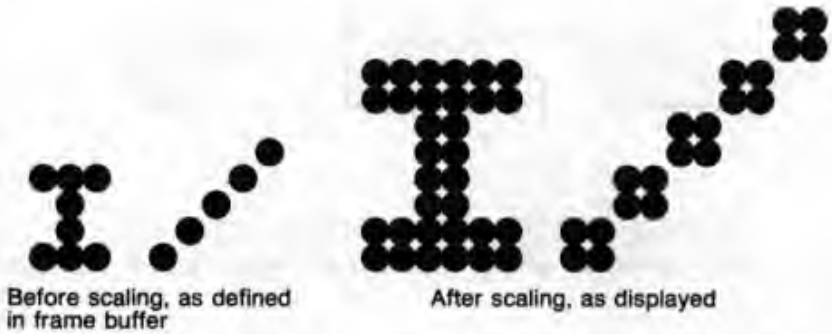


Fig. 4.28 Effect of scaling up a pixmap by a factor of 2.

value is used four times, twice on each of two successive scan lines. Figure 4.28 shows the effect of scaling up a letter and adjoining line by a factor of 2. Unless the image's storage has higher resolution than does its display, scaling up does not reveal more detail: the image is enlarged but has a more jagged appearance. Thus, this animation effect sacrifices spatial resolution but maintains a full range of colors, whereas the double-buffering described in the previous section maintains spatial resolution but decreases the number of colors available in any one image.

In a more general application of image transformations, the scaled image covers only the part of the view surface defined by a viewport, as in Fig. 4.29. Now we must define to the system what is to appear on the view surface outside of the viewport. One possibility is to display some constant color or intensity; another, shown in the figure, is to display the frame buffer itself. The hardware implementation for the latter option is simple. Registers containing the viewport boundary coordinates are compared to the X, Y registers defining the raster scan's current position. If the beam is in the viewport, pixels are fetched from within the window area of the frame buffer and are replicated as required. Otherwise, pixels are fetched from the position in the frame buffer with the same (x, y) coordinates as the beam.

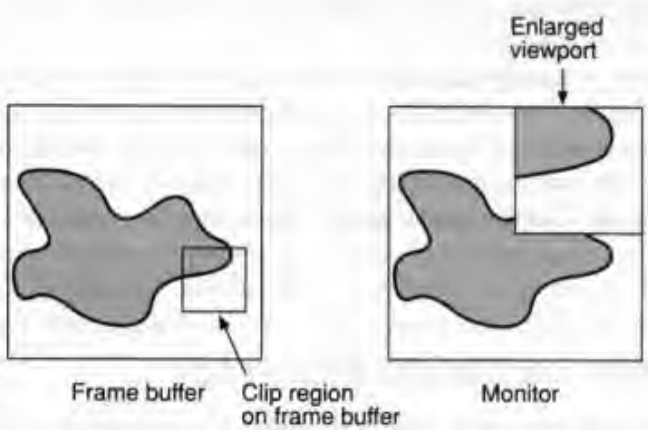


Fig. 4.29 A portion of the frame buffer, specified by the clip region on the frame buffer, scaled up by a factor of 2 and overlaid on the unscaled frame buffer.

Samsung/Google, EX1027, p. 202

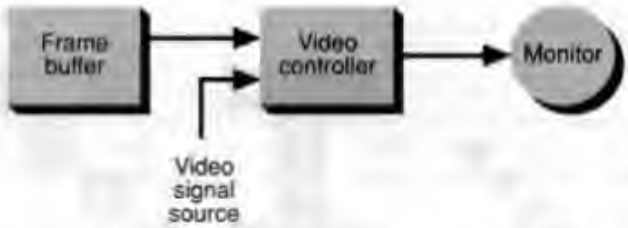


Fig. 4.30 A video controller mixing images from frame buffer and video-signal source.

There are VLSI chips that implement many of these image display functions. These various transformations are special cases of general window-manager operations that might be desired. Window-manager chips are available for use in the video controller [SHIR86]. Each window has a separate pixmap in the system memory, and a data structure in the memory defines window sizes and origins. As a scan line is displayed, the chip knows which window is visible and hence knows from which pixmap to fetch pixels. These and other more advanced hardware issues are discussed in Chapter 18.

4.4.3 Video Mixing

Another useful video-controller function is video mixing. Two images, one defined in the frame buffer and the other defined by a video signal coming from a television camera, recorder, or other source, can be merged to form a composite image. Examples of this merging are seen regularly on television news, sports, and weather shows. Figure 4.30 shows the generic system organization.

There are two types of mixing. In one, a graphics image is set into a video image. The chart or graph displayed over the shoulder of a newscaster is typical of this style. The mixing is accomplished with hardware that treats a designated pixel value in the frame buffer as a flag to indicate that the video signal should be shown instead of the signal from the frame buffer. Normally, the designated pixel value corresponds to the background color of the frame-buffer image, although interesting effects can be achieved by using some other pixel value instead.

The second type of mixing places the video image on top of the frame-buffer image, as when a weather reporter stands in front of a full-screen weather map. The reporter is actually standing in front of a backdrop, whose color (typically blue) is used to control the mixing: Whenever the incoming video is blue, the frame buffer is shown; otherwise, the video image is shown. This technique works well as long as the reporter is not wearing a blue tie or shirt!

4.5 RANDOM-SCAN DISPLAY PROCESSOR

Figure 4.31 shows a typical random (vector) display system. It is generically similar to the display-processor–based raster system architecture discussed in Section 4.3.2. There is of course no pixmap for refreshing the display, and the display processor has no local memory

Samsung/Google, EX1027, p. 203

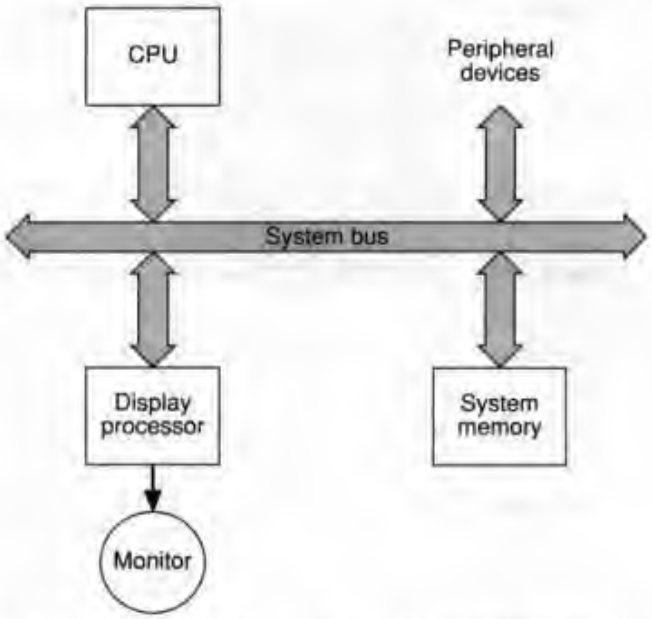


Fig. 4.31 Architecture of a random display system.

for scan-conversion algorithms, since that functionality is typically implemented using programmed logic arrays or microcode.

The random-scan graphics-display processor is often called a *display processing unit* (DPU), or a *graphics controller*. The DPU has an instruction set and instruction address register, and goes through the classic instruction fetch, decode, execute cycle found in any computer. Because there is no pixmap, the display processor must execute its program 30 to 60 times per second in order to provide a flicker-free display. The program executed by the DPU is in main memory, which is shared by the general CPU and the DPU.

The application program and graphics subroutine package also reside in main memory and execute on the general CPU. The graphics package creates a display program of DPU instructions and tells the DPU where to start the program. The DPU then asynchronously executes the display program until it is told to stop by the graphics package. A JUMP instruction at the end of the display program transfers control back to its start, so that the display continues to be refreshed without CPU intervention.

Figure 4.32 shows a set of instructions and mnemonics for a simple random-scan DPU. The processor has X and Y registers and an instruction counter. The instructions are defined for a 16-bit word length. The R/A (relative/absolute) modifier on the LD instructions indicates whether the following coordinate is to be treated as an 11-bit relative coordinate or a 10-bit absolute coordinate. In the former case, the 11 bits are added to either the X or the Y register; in the latter, the 10 bits replace the register contents. (Eleven bits are needed for relative moves for a 2's-complement representation of values in the range -1024 to $+1023$.)

The same R/A interpretation is used for the JUMP instruction, except the instruction counter is modified, thereby effecting a change in the flow of control. The SM, SP, and SL instructions afford compact representation of contour lines, scatter plots, and the like.

Samsung/Google, EX1027, p. 204

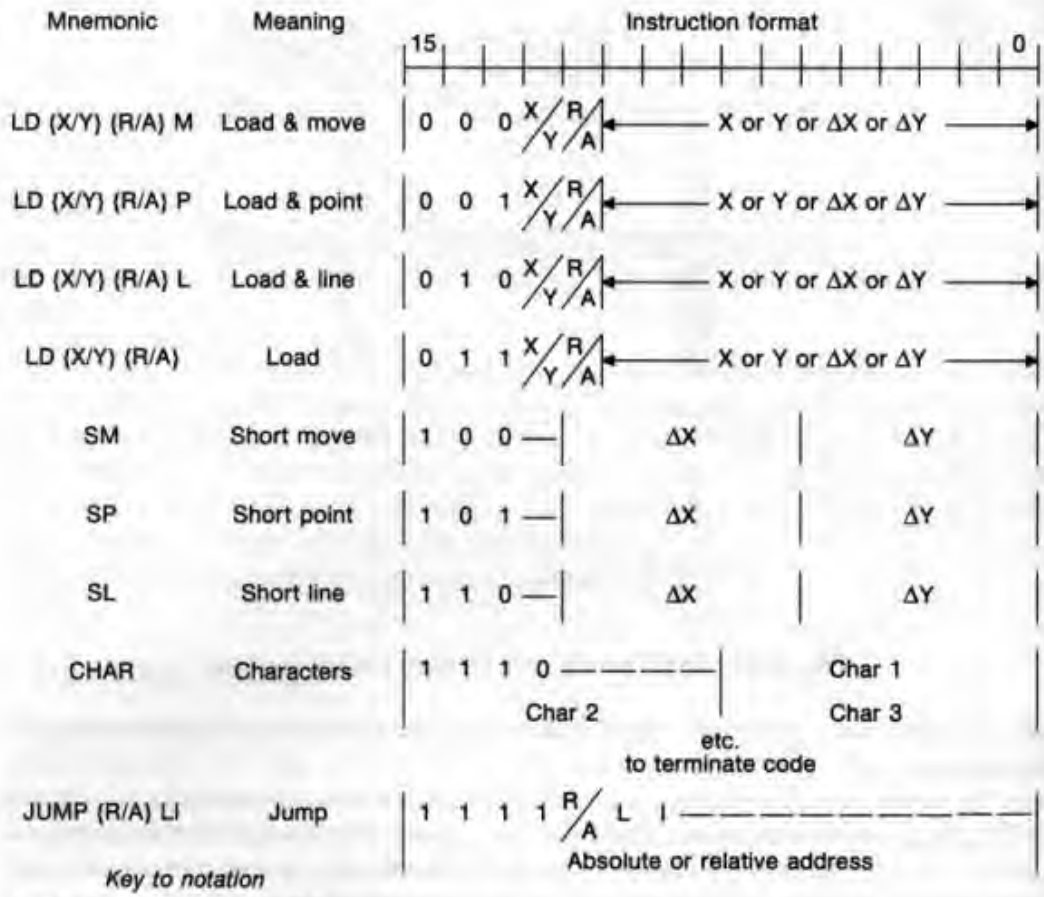


Fig. 4.32 Instruction set for the random-scan display system.

Figure 4.33 is a simple DPU program, written in assembler style, that uses many of these instructions. Notice how the square and diamond are drawn: The first move instruction is absolute, but the rest are relative, so as to facilitate dragging the objects around the screen. If relative instructions were not used, then dragging would require modifying the coordinates of all the instructions used to display the object. The final instruction jumps back to the start of the DPU program. Because the frame lock bit is set, as indicated by the L modifier, execution of the jump is delayed until the next tick of a 30-Hz clock in order to allow the DPU to refresh at 30 Hz but to prevent more frequent refreshing of small programs, which could burn the phosphor.

Notice that, with this instruction set, only one load command (mnemonic LD) is needed to draw or move either horizontally or vertically, because the other coordinate is held constant; oblique movements, however, require two load commands. The two loads can be in either x-then-y or y-then-x order, with the second of the two always specifying a

Samsung/Google, EX1027, p. 205




```
SQUARE:   LDXA      100              Get ready for square
          LDYAM     100              Move to (100, 100)
          LDXRL     800              Line to (900, 100)
          LDYRL     700              Line to (900, 800)
          LDXRL    -800              Line to (100, 800)
          LDYRL    -700              Line to (100, 100), the starting point
POINT:    LDXA      300                                  for square
          LDYAP     450              Point at (300, 450)
DIAMOND:  LDXA      100
          LDYAM     450              Move to (100, 450)
          LDXR      400
          LDYRL    -350              Line to (500, 100)
          LDXR      400
          LDYRL     350              Line to (900, 450)
          LDYR      350
          LDXRL    -400              Line to (500, 800)
          LDXR     -400
          LDYRL    -350              Line to (100, 450), the starting point
TEXT:     LDXA      200                                  for diamond
          LDYAM     900              Move to (200, 900) for text
          CHAR      'DEMONSTRATION t' t is terminate code
          JUMPRL    SQUARE           Regenerate picture, frame lock
```

Fig. 4.33 Program for the random-scan display processor.

move, draw-point, or draw-line operation. For the character command (mnemonic CHAR), a character string follows, ended by a nondisplayable termination code.

There are two main differences between these DPU instructions and the instruction set for a general-purpose computer. With the exception of the JUMP command, all the instructions here are rather special-purpose. A register can be loaded and added to, but the result cannot be stored; the register controls only the position of the CRT beam. The second difference, again excluding JUMP, is that all data are immediate; that is, they are part of the instruction. LDXA 100 means "load the data value 100 into the X register," not "load the contents of address 100 into the X register," as in a computer instruction. This restriction is removed in some of the more advanced DPUs described in Chapter 18.

There are only a few differences between the instruction sets for typical random and raster display processors. The random processor lacks area-filling, bit-manipulation, and look-up table commands. But because of the instruction counter, the random processor does have transfer of control commands. Random displays can work at higher resolutions than can raster displays and can draw smooth lines lacking the jagged edges found on raster displays. The fastest random displays can draw about 100,000 short vectors in a refresh cycle, allowing real-time animation of extremely complex shapes.

Samsung/Google, EX1027, p. 206

4.6 INPUT DEVICES FOR OPERATOR INTERACTION

In this section, we describe the workings of the most common input devices. We present a brief and high-level discussion of how the types of devices available work. In Chapter 8, we discuss the advantages and disadvantages of the various devices, and also describe some more advanced devices.

Our presentation is organized around the concept of *logical devices*, introduced in Chapter 2 as part of SRGP and discussed further in Chapters 7 and 8. There are five basic logical devices: the *locator*, to indicate a position or orientation; the *pick*, to select a displayed entity; the *valuator*, to input a single real number; the *keyboard*, to input a character string; and the *choice*, to select from a set of possible actions or choices. The logical-device concept defines equivalence classes of devices on the basis of the type of information the devices provide to the application program.

4.6.1 Locator Devices

Tablet. A *tablet* (or *data tablet*) is a flat surface, ranging in size from about 6 by 6 inches up to 48 by 72 inches or more, which can detect the position of a movable stylus or puck held in the user's hand. Figure 4.34 shows a small tablet with both a stylus and puck (hereafter, we generally refer only to a stylus, although the discussion is relevant to either). Most tablets use an electrical sensing mechanism to determine the position of the stylus. In one such arrangement, a grid of wires on $\frac{1}{4}$- to $\frac{1}{2}$-inch centers is embedded in the tablet surface. Electromagnetic signals generated by electrical pulses applied in sequence to the wires in the grid induce an electrical signal in a wire coil in the stylus. The strength of the



Fig. 4.34 A data tablet with both a stylus and a puck. The stylus has a pressure-sensitive switch in the tip, which closes when the stylus is pressed. The puck has several pushbuttons for command entry, and a cross-hair cursor for accuracy in digitizing drawings that are placed on the tablet. (Courtesy of Summagraphics Corporation.)

Samsung/Google, EX1027, p. 207

signal induced by each pulse is used to determine the position of the stylus. The signal strength is also used to determine roughly how far the stylus or cursor is from the tablet ("far," "near," (i.e., within about $\frac{1}{2}$ inch of the tablet), or "touching"). When the answer is "near" or "touching," a cursor is usually shown on the display to provide visual feedback to the user. A signal is sent to the computer when the stylus tip is pressed against the tablet, or when any button on the puck (pucks have up to 16 buttons) is pressed.

The tablet's (x, y) position, button status, and nearness state (if the nearness state is "far," then no (x, y) position is available) is normally obtained 30 to 60 times per second. Some tablets can generate an interrupt when any of the following events occur:

- t units of time have passed (thus, the tablet serves as a clock to trigger the updating of the cursor position).
- The puck or stylus has moved more than some distance d. This distance-interval sampling is useful in digitizing drawings in order to avoid recording an excessive number of points.
- The nearness state has changed (some tablets do not report a change from "near" to "far").
- The tip switch or a puck button has been pressed or released. It is important to have available both *button-down* and *button-up* events from the tablet and other such devices. For instance, a button-down event might cause a displayed object to start growing larger, and the button-up event then might stop the size change.
- The cursor enters a specified rectangular area of the tablet surface.

Relevant parameters of tablets and other locator devices are their resolution (number of distinguishable points per inch), linearity, repeatability, and size or range. These parameters are particularly crucial for digitizing maps and drawings; they are of less concern when the device is used only to position a screen cursor, because the user has the feedback of the screen cursor position to guide his hand movements, and because the resolution of a typical display is much less than that of even inexpensive tablets.

Other tablet technologies use sound (sonic) coupling and resistive coupling. The *sonic tablet* uses sound waves to couple the stylus to microphones positioned on the periphery of the digitizing area. Sound bursts are created by an electrical spark at the tip of the stylus. The delay between when the spark occurs and when its sound arrives at each microphone is proportional to the distance from the stylus to each microphone (see Fig. 4.35). The sonic tablet is advantageous for digitizing drawings bound in thick books, where a normal tablet stylus does not get close enough to the tablet to record an accurate position. Also, it does not require a dedicated working area, as the other tablets do. Sonic coupling is also useful in 3D positioning devices, as discussed in Chapter 8.

One type of *resistive tablet* uses a battery-powered stylus that emits high-frequency radio signals. The tablet, which is just a piece of glass, is coated with a thin layer of conducting material in which an electrical potential is induced by the radio signals. The strength of the signals at the edges of the tablet is inversely proportional to the distance to the stylus and can thus be used to calculate the stylus position. Another similar tablet uses a resistive polymer mesh stretched across the faceplate of a CRT [SUNF86]. The stylus

Samsung/Google, EX1027, p. 208

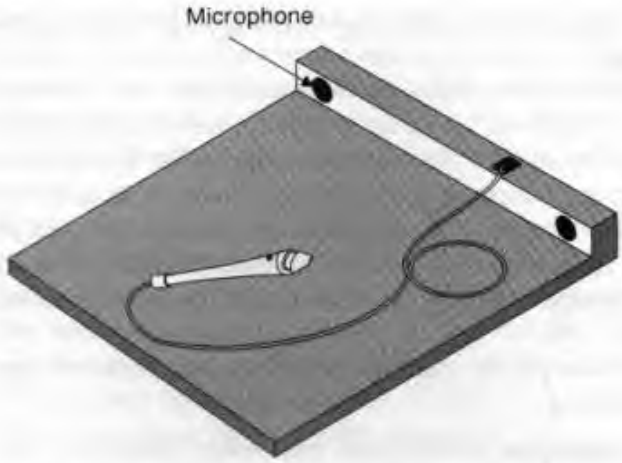


Fig. 4.35 A 2D sound tablet. Sound waves emitted from the stylus are received by the two microphones at the rear of the tablet.

applies an electrical potential to the mesh, and the stylus's position is determined from the voltage drop across the mesh.

Most tablet styluses have to be connected to the tablet controller by a wire. The stylus of the resistive tablet can be battery-powered, as shown in Fig. 4.36. Such a stylus is attractive because there is no cord to get in the way. On the other hand, walking away with it in your pocket is easy to do!

Several types of tablets are transparent, and thus can be back-lit for digitizing X-ray films and photographic negatives, and can also be mounted directly over a CRT. The resistive tablet is especially well suited for this, as it can be curved to the shape of the CRT.



Fig. 4.36 The Penmouse tablet with its cordless three-button, battery-powered stylus and puck. (Courtesy of Kurta Corporation.)

Samsung/Google, EX1027, p. 209

Mouse. A mouse is a small hand-held device whose relative motion across a surface can be measured. Mice differ in the number of buttons and in how relative motion is detected. Other important differences between various types of mice are discussed in Section 8.1.7. The motion of the roller in the base of a *mechanical mouse* is converted to digital values that are used to determine the direction and magnitude of movement. The *optical mouse* is used on a special pad having a grid of alternating light and dark lines. A light-emitting diode (LED) on the bottom of the mouse directs a beam of light down onto the pad, from which it is reflected and sensed by detectors on the bottom of the mouse. As the mouse is moved, the reflected light beam is broken each time a dark line is crossed. The number of pulses so generated, which is equal to the number of lines crossed, are used to report mouse movements to the computer.

Because mice are relative devices, they can be picked up, moved, and then put down again without any change in reported position. (A series of such movements is often called "stroking" the mouse.) The relative nature of the mouse means that the computer must maintain a "current mouse position," which is incremented or decremented by mouse movements.

Trackball. The *trackball*, one of which is shown in Fig. 4.37, is often described as an upside-down mechanical mouse. The motion of the trackball, which rotates freely within its housing, is sensed by potentiometers or shaft encoders. The user typically rotates the trackball by drawing the palm of her hand across the ball. Various switches are usually mounted within finger reach of the trackball itself and are used in ways analogous to the use of mouse and tablet-puck buttons.

Joystick. The *joystick* (Fig. 4.38) can be moved left or right, forward or backward; again, potentiometers sense the movements. Springs are often used to return the joystick to its home center position. Some joysticks, including the one pictured, have a third degree of freedom: the stick can be twisted clockwise and counterclockwise. The *isometric joystick*,



Fig. 4.37 Trackball with several nearby switches. (Courtesy of Measurement Systems, Inc.)

Samsung/Google, EX1027, p. 210



Fig. 4.38 A joystick with a third degree of freedom. The joystick can be twisted clockwise and counterclockwise. (Courtesy of Measurement Systems, Inc.)

shown in Fig. 4.39, is rigid: strain gauges on the shaft measure slight deflections caused by force applied to the shaft.

It is difficult to use a joystick to control the absolute position of a screen cursor directly, because a slight movement of the (usually) short shaft is amplified five or ten times in the movement of the cursor. This makes the screen cursor's movements quite jerky and does not allow quick and accurate fine positioning. Thus, the joystick is often used to control the velocity of the cursor movement rather than the absolute cursor position. This means that the current position of the screen cursor is changed at rates determined by the joystick.

The *joyswitch*, a variant of the joystick, is found in some home and arcade computer games. The stick can be moved in any of eight directions: up, down, left, right, and the four



Fig. 4.39 An isometric joystick. (Courtesy of Measurement Systems, Inc.)

Samsung/Google, EX1027, p. 211

diagonal directions. Small switches sense in which of the eight directions the stick is being pushed.

Touch panel. Mice, trackballs, and joysticks all take up work-surface area. The *touch panel* allows the user to point at the screen directly with a finger to move the cursor around on the screen. Several different technologies are used for touch panels. Low-resolution panels (from 10 to 50 resolvable positions in each direction) use a series of infrared LEDs and light sensors (photodiodes or phototransistors) to form a grid of invisible light beams over the display area. Touching the screen breaks one or two vertical and horizontal light beams, thereby indicating the finger's position. If two parallel beams are broken, the finger is presumed to be centered between them; if one is broken, the finger is presumed to be on the beam.

A capacitively coupled touch panel can provide about 100 resolvable positions in each direction. When the user touches the conductively coated glass panel, electronic circuits detect the touch position from the impedance change across the conductive coating [INTE85].

One high-resolution panel (about 500 resolvable positions in each direction) uses sonar-style ranging. Bursts of high-frequency sound waves traveling alternately horizontally and vertically are introduced at the edge of a glass plate. The touch of a finger on the glass causes part of the wave to be reflected back to its source. The distance to the finger can be calculated from the interval between emission of the wave burst and its arrival back at the source. Another high-resolution panel uses two slightly separated layers of transparent material, one coated with a thin layer of conducting material and the other with resistive material. Fingertip pressure forces the layers to touch, and the voltage drop across the resistive substrate is measured and is used to calculate the coordinates of the touched position. Low-resolution variations of this method use bars of conducting material or thin wires embedded in the transparent material. Touch-panel technologies have been used to make small positioning pads for keyboards.

The most significant touch-panel parameters are resolution, the amount of pressure required for activation (not an issue for the light-beam panel), and transparency (again, not an issue for the light-beam panel). An important issue with some of the technologies is parallax: If the panel is $\frac{1}{2}$ inch away from the display, then users touch the position on the panel that is aligned with their eyes and the desired point on the display, not at the position on the panel directly perpendicular to the desired point on the display.

Users are accustomed to some type of tactile feedback, but touch panels of course offer none. It is thus especially important that other forms of immediate feedback be provided, such as an audible tone or highlighting of the designated target or position.

Light pen. *Light pens* were developed early in the history of interactive computer graphics. The pen is misnamed; it *detects* light pulses, rather than emitting light as its name implies. The event caused when the light pen sees a light pulse on a raster display can be used to save the video controller's X and Y registers and interrupt the computer. By reading the saved values, the graphics package can determine the coordinates of the pixel seen by the light pen. The light pen cannot report the coordinates of a point that is completely black, so special techniques are needed to use the light pen to indicate an arbitrary position: one is to display, for a single frame time, a dark blue field in place of the regular image.

The light pen, when used with a vector display, acts as a pick rather than a positioning

Samsung/Google, EX1027, p. 212

device. When the light pen senses light, the DPU is stopped and the CPU is interrupted. The CPU then reads the DPU's instruction address register, the contents of which can be used to determine which output primitive was being drawn when the interrupt occurred. As with the raster display, a special technique, called *light-pen tracking*, is needed to indicate positions within a single vector with the light pen [FOLE82].

The light pen is an aging technology with limited use. Unless properly adjusted, light pens sometimes detect false targets, such as fluorescent lights or other nearby graphics primitives (e.g., adjacent characters), and fail to detect intended targets. When used over several hours, a light pen can be tiring for inexperienced users, because it must be picked up, pointed, and set down for each use.

4.6.2 Keyboard Devices

The *alphanumeric keyboard* is the prototypical text input device. Several different technologies are used to detect a key depression, including mechanical contact closure, change in capacitance, and magnetic coupling. The important functional characteristic of a keyboard device is that it creates a code (ASCII, EBCDIC, etc.) uniquely corresponding to a pressed key. It is sometimes desirable to allow *chording* (pressing several keys at once) on an alphanumeric keyboard, to give experienced users rapid access to many different commands. This is in general not possible with the standard *coded keyboard*, which returns an ASCII code per keystroke and returns nothing if two keys are pressed simultaneously (unless the additional keys were shift, control, or other special keys). In contrast, an *unencoded keyboard* returns the identity of all keys that are pressed simultaneously, thereby allowing chording.

4.6.3 Valuator Devices

Most valuator devices that provide scalar values are based on potentiometers, like the volume and tone controls of a stereo set. Valuators are usually rotary potentiometers (dials), typically mounted in a group of eight or ten, as in Fig. 4.40. Simple rotary potentiometers



Fig. 4.40 A bank of eight rotary potentiometers. The readouts on the light-emitting diodes (LED) above each dial can be used to label each dial, or to give the current setting. (Courtesy of Evans and Sutherland Computer Corporation.)

Samsung/Google, EX1027, p. 213



Fig. 4.41 Function keys with light-emitting diode (LED) labels, integrated with a keyboard unit. (Courtesy of Evans and Sutherland Computer Corporation.)

can be rotated through about 330°; this may not be enough to provide both adequate range and resolution. Continuous-turn potentiometers can be rotated freely in either direction, and hence are unbounded in range. Linear potentiometers, which are of necessity bounded devices, are used infrequently in graphics systems.

4.6.4 Choice Devices

Function keys are the most common choice device. They are sometimes built as a separate unit, but more often are integrated with a keyboard. Other choice devices are the *buttons* found on many tablet pucks and on the mouse. Choice devices are generally used to enter commands or menu options a graphics program. Dedicated-purpose systems can use function keys with permanent key-cap labels. So that labels can be changeable or "soft," function keys can include a small LCD or LED display next to each button or in the key caps themselves, as shown in Fig. 4.41. Yet another alternative is to place buttons on the bezel of the display, so that button labels can be shown on the display itself, right next to the physical button.

4.7 IMAGE SCANNERS

Although data tablets can be used to digitize existing line drawings manually, this is a slow and tedious process, unsuitable for more than a few simple drawings—and it does not work at all for half-tone images. Image scanners provide an efficient solution. A television camera used in conjunction with a digital frame grabber is an inexpensive way to obtain moderate-resolution (1000 by 1000, with multiple intensity levels) raster images of black-and-white or color photographs. Slow-scan charge-coupled–device (CCD) television cameras can create a 2000 by 2000 image in about 30 seconds. An even lower-cost approach uses a scan head, consisting of a grid of light-sensing cells, mounted on the print head of a printer; it scans images at a resolution of about 80 units per inch. These resolutions are not acceptable for high-quality publication work, however. In such cases, a *photo scanner* is used. The photograph is mounted on a rotating drum. A finely collimated light beam is directed at the photo, and the amount of light reflected is measured by a photocell. For a negative, transmitted light is measured by a photocell inside the drum.

Samsung/Google, EX1027, p. 214

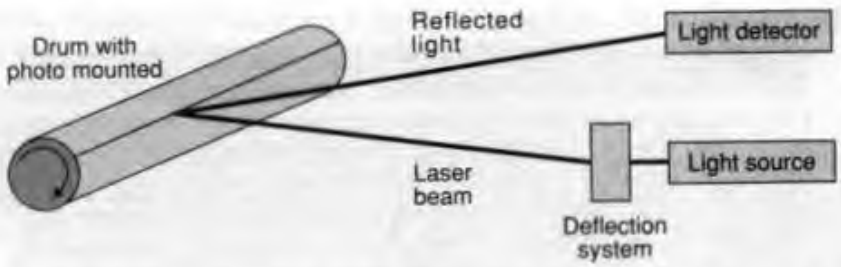


Fig. 4.42 A photo scanner. The light source is deflected along the drum axis, and the amount of reflected light is measured.

which is transparent. As the drum rotates, the light source slowly moves from one end to the other, thus doing a raster scan of the entire photograph (Fig. 4.42). For colored photographs, multiple passes are made, using filters in front of the photocell to separate out various colors. The highest-resolution scanners use laser light sources, and have resolutions greater then 2000 units per inch.

Another class of scanner uses a long thin strip of CCDs, called a *CCD array*. A drawing is digitized by passing it under the CCD array, incrementing the drawing's movement by whatever resolution is required. Thus, a single pass, taking 1 or 2 minutes, is sufficient to digitize a large drawing. Resolution of the CCD array is 200 to 1000 units per inch, which is less than that of the photo scanner. Such a scanner is shown in Fig. 4.43.

Line drawings can easily be scanned using any of the approaches we have described. The difficult part is distilling some meaning from the collection of pixels that results. *Vectorizing* is the process of extracting lines, characters, and other geometric primitives



Fig. 4.43 A liquid-crystal display (LCD) scanner. The lens and linear LCD array in the scanning head (top) move left to right to scan a photo or object. (Photographs provided by Truvel Corporation.)

Samsung/Google, EX1027, p. 215

from a raster image. This task requires appropriate algorithms, not scanning hardware, and is essentially an image-processing problem involving several steps. First, thresholding and edge enhancement are used to clean up the raster image—to eliminate smudges and smears and to fill in gaps. Feature-extraction algorithms are then used to combine adjacent "on" pixels into geometric primitives such as straight lines. At a second level of complexity, pattern-recognition algorithms are used to combine the simple primitives into arcs, letters, symbols, and so on. User interaction may be necessary to resolve ambiguities caused by breaks in lines, dark smudges, and multiple lines intersecting near one another.

A more difficult problem is organizing a collection of geometric primitives into meaningful data structures. A disorganized collection of lines is not particularly useful as input to a CAD or topographic (mapping) application program. The higher-level geometric constructs represented in the drawings need to be recognized. Thus, the lines defining the outline of a county should be organized into a polygon primitive, and the small "+" representing the center of an arc should be grouped with the arc itself. There are partial solutions to these problems. Commercial systems depend on user intervention when the going gets tough, although algorithms are improving continually.

EXERCISES

4.1 For an electrostatic plotter with 18-inch-wide paper, a resolution of 200 units to the inch in each direction, and a paper speed of 3 inches per second, how many bits per second must be provided to allow the paper to move at full speed?

4.2 If long-persistence phosphors decrease the fusion frequency, why not use them routinely?

4.3 Write a program to display test patterns on a raster display. Three different patterns should be provided: (1) horizontal lines 1 pixel wide, separated by 0, 1, 2, or 3 pixels; (2) vertical lines 1 pixel wide, separated by 0, 1, 2, or 3 pixels; and (3) a grid of 1-pixel dots on a grid spaced at 5-pixel intervals. Each pattern should be displayable in white, red, green, or blue, as well as alternating color bars. How does what you observe when the patterns are displayed relate to the discussion of raster resolution?

4.4 Prepare a report on technologies for large-screen displays.

4.5 How long would it take to load a 512 by 512 by 1 bitmap, assuming that the pixels are packed 8 to a byte and that bytes can be transferred and unpacked at the rate of 100,000 bytes per second? How long would it take to load a 1024 by 1280 by 1 bitmap?

4.6 Design the logic of a hardware unit to convert from 2D raster addresses to byte plus bit-within-byte addresses. The inputs to the unit are as follows: (1) (x, y), a raster address; (2) *base*, the address of the memory byte that has raster address (0, 0) in bit 0; and (3) x_{max}, the maximum raster x address (0 is the minimum). The outputs from the unit are as follows: (1) *byte*, the address of the byte that contains (x, y) in one of its bits; and (2) *bit*, the bit within *byte* which contains (x, y). What simplifications are possible if $x_{max} + 1$ is a power of 2?

4.7 Program a raster copy operation that goes from a bitmap into a virtual-memory area that is scattered across several pages. The hardware raster copy on your display works only in physical address space, so you must set up a scatter-write situation, with multiple moves invoked, one for each logical page contained in the destination.

4.8 Design an efficient instruction-set encoding for the simple raster display instruction set of Section 4.3.2. By "efficient" is meant "minimizing the number of bits used per instruction."

Samsung/Google, EX1027, p. 216

4.9 Using the instruction set for the simple raster display of Section 4.3.2, write program segments to do the following.

a. Double-buffer a moving object. You need not actually write the code that moves and draws the object; just indicate its placement in the overall code sequence.

b. Draw a pie chart, given as input a list of data values. Use different colors for each slice of the pie, choosing colors you believe will look good together.

c. Animate the left-to-right movement of circles arranged in a horizontal row. Each circle is five units in radius and the circles are placed on 15-unit centers. Use the animation technique discussed in Section 4.4.1. At each step in the animation, every fourth circle should be visible.

d. Write a program that uses RasterOp to drag a 25 by 25 icon around on the screen, using an offscreen area of the bitmap to store both the icon and the part of the bitmap currently obscured by the icon.

4.10 In a raster scan of n lines with m pixels per line displayed at r cycles per second, there is a certain amount of time during which no image is being traced: the horizontal retrace time t_h, which occurs once per scan line, and the vertical retrace time t_v, which occurs once per frame.

a. Derive equations for the percentage of time that no image is being traced, remembering that different equations are needed for the interlaced and noninterlaced cases, because the two fields per frame of an interlaced scan each require a vertical retrace time.

b. Evaluate the equation for the following values, taken from [WHIT84]:

Visible area pixels × lines	Refresh rate, Hz	Interlace	Vertical retrace time, microseconds	Horizontal retrace time, microseconds
640 × 485	30	yes	1271	11
1280 × 1024	30	yes	1250	7
640 × 485	60	no	1250	7
1280 × 1024	60	no	600	4

c. What is the meaning of the percentages you have calculated?

4.11 Develop a design strategy for a video controller that can display a 960 by x image on a 480 by $x/2$ display, where x is the horizontal resolution. (This is the type of device needed to drive a video recorder from a high-resolution display system.)

4.12 A raster display has 4 bits per pixel and a look-up table with 12 bits per entry (4 bits each for red, green, and blue). Think of the four planes of the pixmap as being partitioned to hold two images: image 1 in the two high-order planes, image 0 in the two low-order planes. The color assignments for each of the 2-bit pixel values in each image are 00 = red, 01 = green, 10 = blue, and 11 = white.

a. Show how to load the look-up table so that just image 1 is displayed.

b. Show how to load the look-up table so that just image 0 is displayed.

Note that these are the look-up tables needed for the two phases of double-buffering, as discussed in Section 4.4.1.

4.13 Given a 16-entry by 12-bit-per-entry look-up table with two 2-bit images, show how it should be loaded to produce the lap-dissolve given by the expression $image1 * t + image0 * (1 - t)$ for values of $t = 0.0, 0.25$, and 0.5. The look-up table color assignments are the same as in Exercise 4.12. (See Chapter 17.)

4.14 In Section 4.4.1, two animation methods are discussed. What are the advantages and disadvantages of each?

Samsung/Google, EX1027, p. 217

4.15 Redesign the simple DPU instruction set of Fig. 4.32 for a 16-bit word, assuming that all opcodes are equal in length.

4.16 Consider means for picking (as with a light pen) a line on a raster display when all that can be detected is a single pixel.

Samsung/Google, EX1027, p. 218

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

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

Samsung/Google, EX1027, p. 219

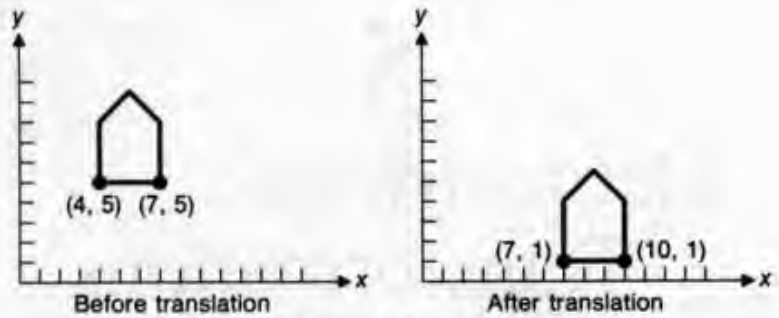


Fig. 5.1 Translation of a house.

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.

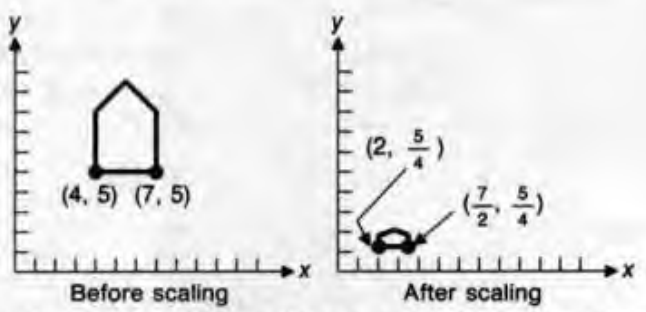


Fig. 5.2 Scaling of a house. The scaling is nonuniform, and the house changes position.

Samsung/Google, EX1027, p. 220

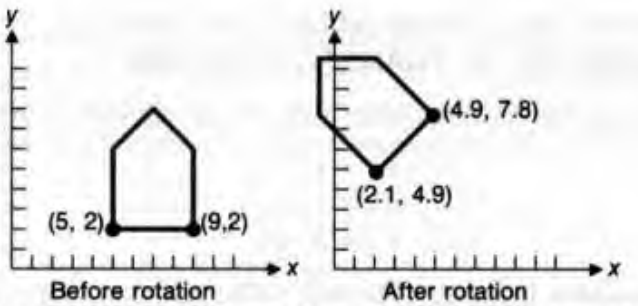


Fig. 5.3 Rotation of a house. The house also changes position.

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).

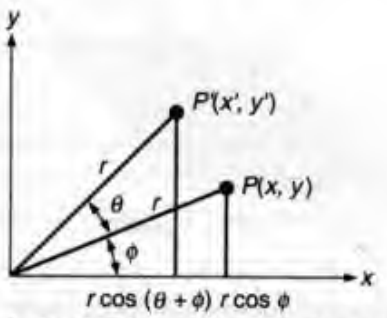


Fig. 5.4 Derivation of the rotation equation.

Samsung/Google, EX1027, p. 221

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

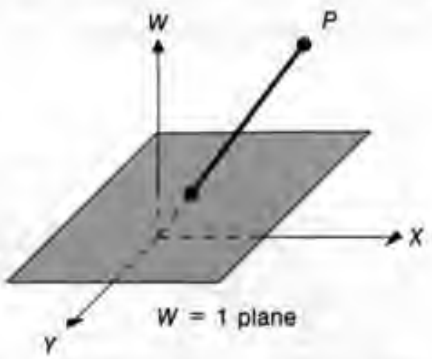


Fig. 5.5 The *XYW* homogeneous coordinate space, with the $W = 1$ plane and point $P(X, Y, W)$ projected onto the $W = 1$ plane.

Samsung/Google, EX1027, p. 222

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:

$$(P \cdot M)^{\mathrm{T}} = M^{\mathrm{T}} \cdot P^{\mathrm{T}}.$$

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

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

Samsung/Google, EX1027, p. 223

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).

Samsung/Google, EX1027, p. 224

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.

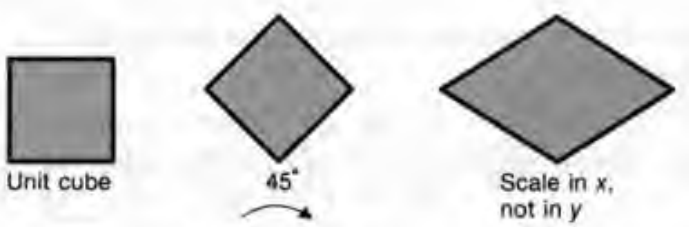


Fig. 5.6 A unit cube is rotated by $-45°$ and is nonuniformly scaled. The result is an affine transformation of the unit cube, in which parallelism of lines is maintained, but neither angles nor lengths are maintained.

Samsung/Google, EX1027, p. 225

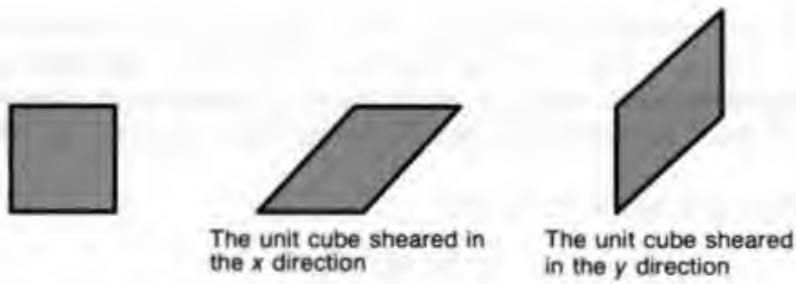


Fig. 5.7 The primitive-shear operations applied to the unit cube. In each case, the lengths of the oblique lines are now greater than 1.

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

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

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

Samsung/Google, EX1027, p. 226

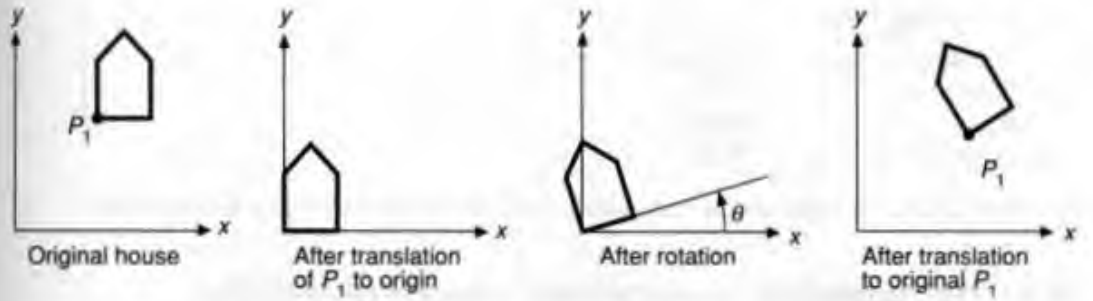


Fig. 5.8 Rotation of a house about the point P_1 by an angle θ.

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

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

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

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:

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

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:

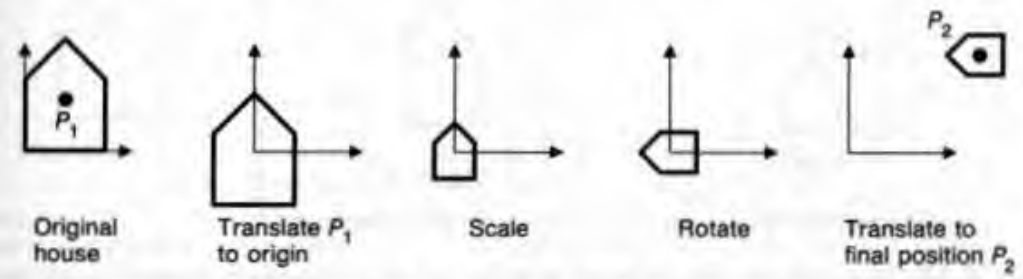


Fig. 5.9 Rotation of a house about the point P_1, and placement such that what was at P_1 is at P_2.

Samsung/Google, EX1027, p. 227

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

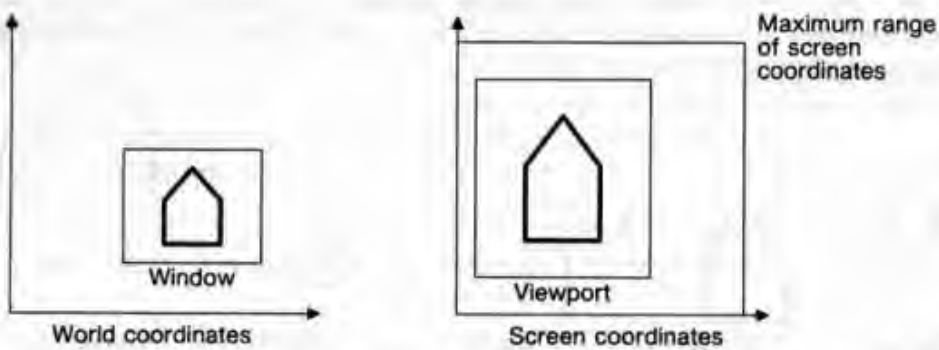


Fig. 5.10 The window in world coordinates and the viewport in screen coordinates determine the mapping that is applied to all the output primitives in world coordinates.

Samsung/Google, EX1027, p. 228

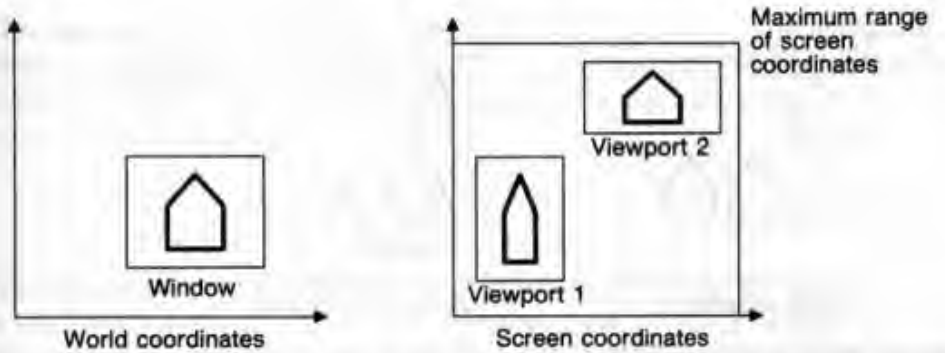


Fig. 5.11 The effect of drawing output primitives with two viewports. Output primitives specifying the house were first drawn with viewport 1, the viewport was changed to viewport 2, and then the application program again called the graphics package to draw the output primitives.

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

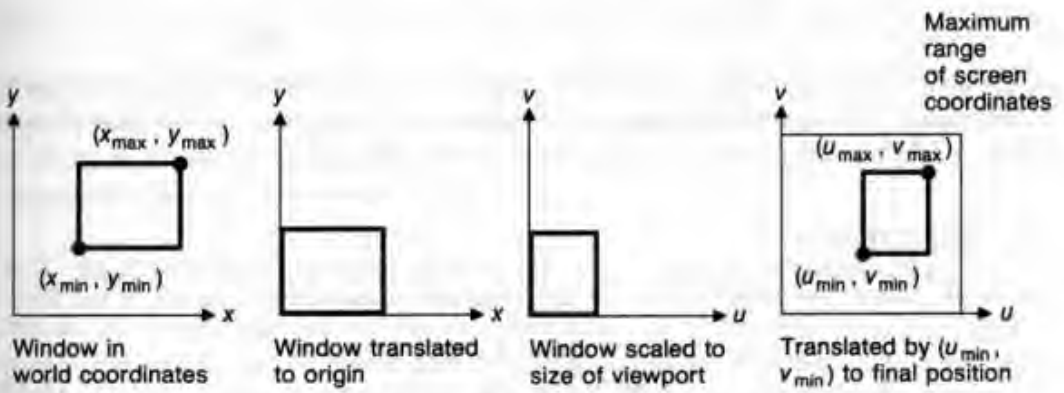


Fig. 5.12 The steps in transforming a world-coordinate window into a viewport.

Samsung/Google, EX1027, p. 229

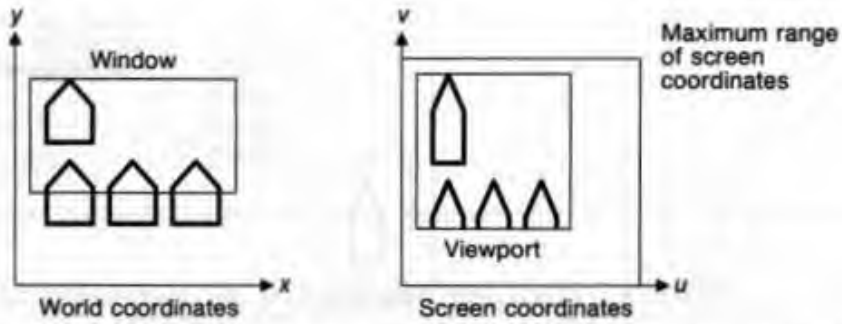


Fig. 5.13 Output primitives in world coordinates are clipped against the window. Those that remain are displayed in the viewport.

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:

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

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

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

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

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

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

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

Samsung/Google, EX1027, p. 230

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

Samsung/Google, EX1027, p. 231

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

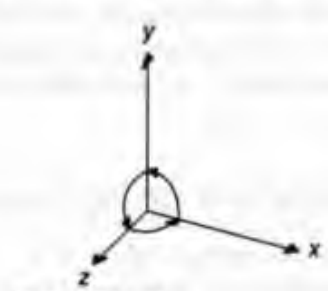


Fig. 5.14 The right-handed coordinate system.

Samsung/Google, EX1027, p. 232

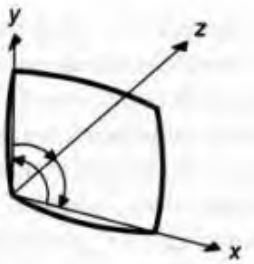


Fig. 5.15 The left-handed coordinate system, with a superimposed display screen.

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

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

Samsung/Google, EX1027, p. 233

The columns (and the rows) of the upper-left 3 × 3 submatrix of $R_x(\theta)$, $R_z(\theta)$, 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

Samsung/Google, EX1027, p. 234

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.



(a) Initial position (b) Final position

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

Samsung/Google, EX1027, p. 235

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

$$\begin{aligned} \cos(\theta - 90) &= \sin\theta = \frac{z'_2}{D_1} = \frac{z_2 - z_1}{D_1}, \\ \sin(\theta - 90) &= -\cos\theta = -\frac{x'_2}{D_1} = -\frac{x_2 - x_1}{D_1}, \end{aligned} \tag{5.53}$$

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.

Samsung/Google, EX1027, p. 236

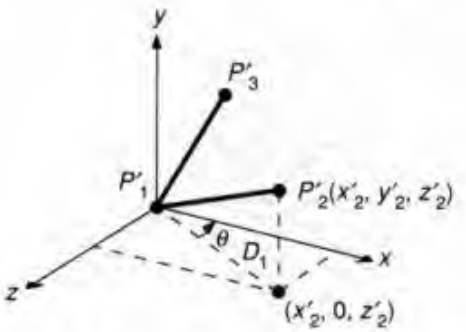


Fig. 5.17 Rotation about the y axis: The projection of $P_1'P_2'$, which has length D_1, is rotated into the z axis. The angle θ shows the positive direction of rotation about the y axis: The actual angle used is $-(90 - \theta)$.

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

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

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

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

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

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

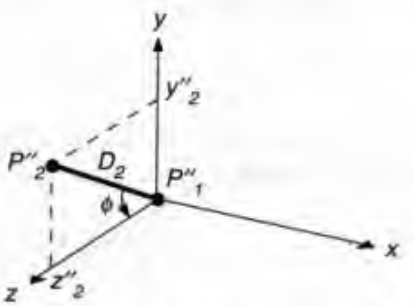


Fig. 5.18 Rotation about the x axis: $P_1''P_2''$ is rotated into the z axis by the positive angle ϕ. D_2 is the length of the line segment. The line segment $P_1''P_3''$ is not shown, because it is not used to determine the angles of rotation. Both lines are rotated by $R_x(\phi)$.

Samsung/Google, EX1027, p. 237

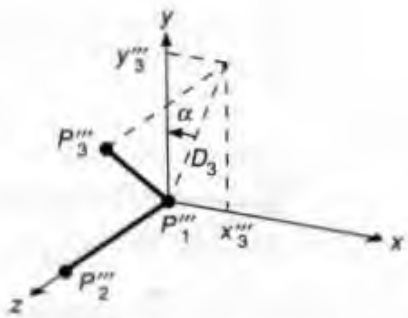


Fig. 5.19 Rotation about the z axis: The projection of $P'_1P'_3$, whose length is D_3, is rotated by the positive angle α into the y axis, bringing the line itself into the (y, z) plane. D_3 is the length of the projection.

The rotation is through the positive angle α, with

$$\cos\alpha = y'''_3/D_3, \quad \sin\alpha = x'''_3/D_3, \quad D_3 = \sqrt{x'''^2_3 + y'''^2_3}. \tag{5.60}$$

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

$$R = \begin{bmatrix} r_{1_x} & r_{2_x} & r_{3_x} \\ r_{1_y} & r_{2_y} & r_{3_y} \\ r_{1_z} & r_{2_z} & r_{3_z} \end{bmatrix}. \tag{5.62}$$

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

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

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:

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

Finally,

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

Samsung/Google, EX1027, p. 238

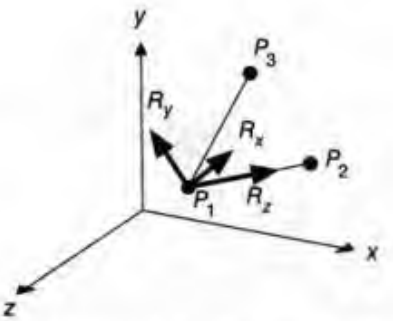


Fig. 5.20 The unit vectors R_x, R_y, and R_z, which are transformed into the principal axes.

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

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

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, 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. 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 $z_p \times x_p = DOF \times (y \times$

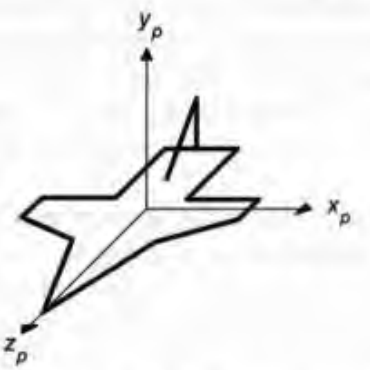


Fig. 5.21 An airplane in the (x_p, y_p, z_p) coordinate system.

Samsung/Google, EX1027, p. 239

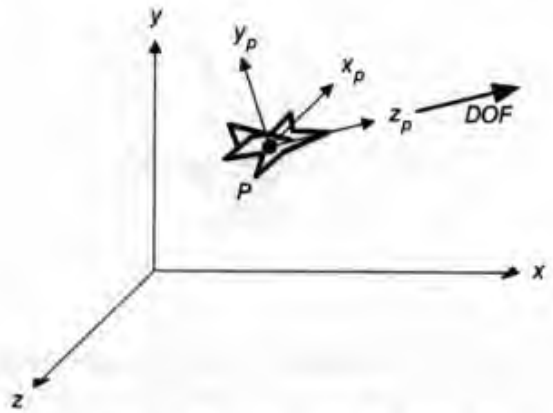


Fig. 5.22 The airplane of Figure 5.21 positioned at point P, and headed in direction *DOF*.

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 $|DOF|$:

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

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,

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

Samsung/Google, EX1027, p. 240

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 $1 \leq i, j \leq 4$.

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

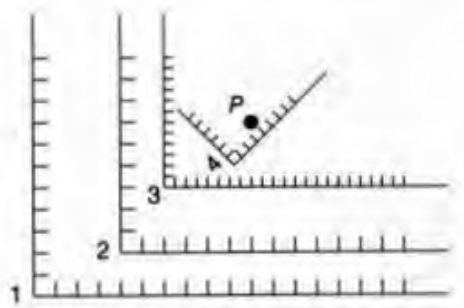


Fig. 5.23 The point P and coordinate systems 1, 2, 3, and 4.

Samsung/Google, EX1027, p. 241

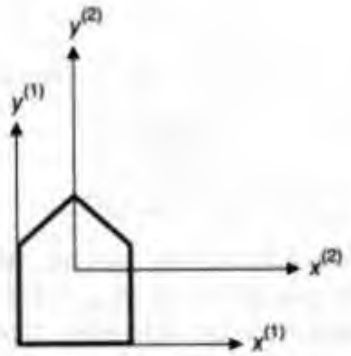


Fig. 5.24 The house and two coordinate systems. Coordinates of points on the house can be represented in either coordinate system.

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, $M_{2\leftarrow1}$—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

$$\begin{aligned} M_{5\leftarrow1} &= M_{5\leftarrow4}\, M_{4\leftarrow3}\, M_{3\leftarrow2}\, M_{2\leftarrow1} \\ &= (T(x_2, y_2) \cdot R(\theta) \cdot S(s_x, s_y) \cdot T(-x_1, -y_1))^{-1} \\ &= T(x_1, y_1) \cdot S(s_x^{-1}, s_y^{-1}) \cdot R(-\theta) \cdot T(-x_2, -y_2), \end{aligned} \tag{5.72}$$

so that

$$P^{(5)} = M_{5\leftarrow1}P^{(1)} = T(x_1, y_1) \cdot S(s_x^{-1}, s_y^{-1}) \cdot R(-\theta) \cdot T(-x_2, -y_2) \cdot P^{(1)}. \tag{5.73}$$

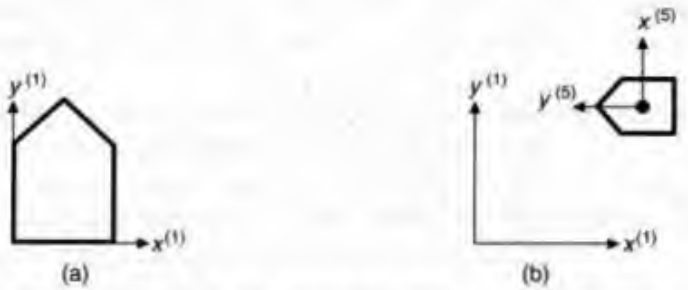


Fig. 5.25 The original house (a) in its coordinate system and the transformed house (b) in its coordinate system with respect to the original coordinate system.

Samsung/Google, EX1027, p. 242

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 $Q^{(i)} \cdot P^{(i)} = M_{i \leftarrow j} \cdot Q^{(j)} \cdot P^{(j)}$. 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

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

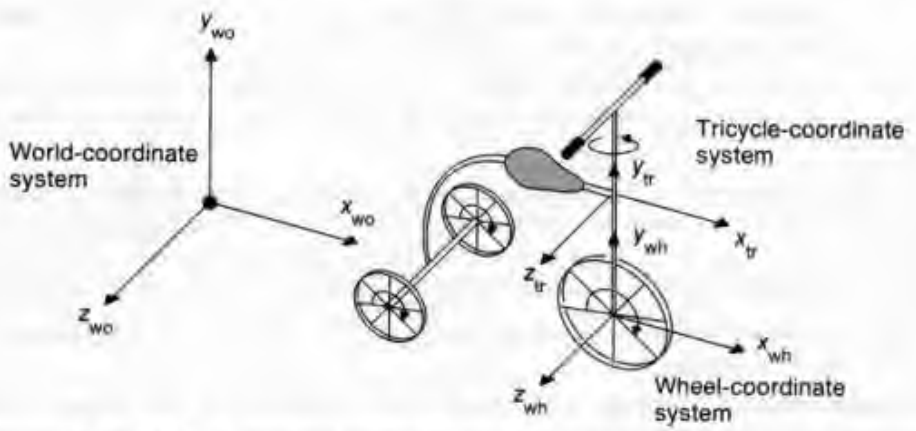


Fig. 5.26 A stylized tricycle with three coordinate systems.

Samsung/Google, EX1027, p. 243

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.

Samsung/Google, EX1027, p. 244

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.

Samsung/Google, EX1027, p. 245

6
Viewing in 3D

The 3D viewing process is inherently more complex than is the 2D viewing process. In 2D, we simply specify a window on the 2D world and a viewport on the 2D view surface. Conceptually, objects in the world are clipped against the window and are then transformed into the viewport for display. The extra complexity of 3D viewing is caused in part by the added dimension and in part by the fact that display devices are only 2D.

The solution to the mismatch between 3D objects and 2D displays is accomplished by introducing *projections*, which transform 3D objects onto a 2D projection plane. Much of this chapter is devoted to projections: what they are, their mathematics, and how they are used in a current graphics subroutine package, PHIGS [ANSI88]. Their use is also discussed further in Chapter 7.

In 3D viewing, we specify a *view volume* in the world, a projection onto a projection plane, and a viewport on the view surface. Conceptually, objects in the 3D world are clipped against the 3D view volume and are then projected. The contents of the projection of the view volume onto the projection plane, called the *window*, are then transformed (mapped) into the viewport for display. Figure 6.1 shows this conceptual model of the 3D viewing process, which is the model presented to the users of numerous 3D graphics subroutine packages. Just as with 2D viewing, a variety of strategies can be used to implement the viewing process. The strategies need not be identical to the conceptual model, so long as the results are those defined by the model. A typical implementation strategy for wire-frame line drawings is described in Section 6.5. For graphics systems that perform visible-surface determination and shading, a somewhat different pipeline, discussed in Chapter 16, is used.

Samsung/Google, EX1027, p. 246

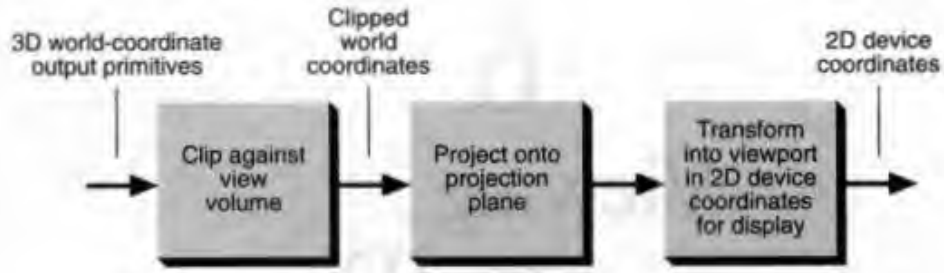


Fig. 6.1 Conceptual model of the 3D viewing process.

6.1 PROJECTIONS

In general, projections transform points in a coordinate system of dimension n into points in a coordinate system of dimension less than n. In fact, computer graphics has long been used for studying n-dimensional objects by projecting them into 2D for viewing [NOLL67]. Here, we shall limit ourselves to the projection from 3D to 2D. The projection of a 3D object is defined by straight projection rays (called *projectors*) emanating from a *center of projection*, passing through each point of the object, and intersecting a *projection plane* to form the projection. Figure 6.2 shows two different projections of the same line. Fortunately, the projection of a line is itself a line, so only line endpoints need actually to be projected.

The class of projections we deal with here is known as *planar geometric projections* because the projection is onto a plane rather than some curved surface and uses straight rather than curved projectors. Many cartographic projections are either nonplanar or nongeometric. Similarly, the Omnimax film format requires a nongeometric projection [MAX82].

Planar geometric projections, hereafter referred to simply as *projections*, can be divided into two basic classes: *perspective* and *parallel*. The distinction is in the relation of the center of projection to the projection plane. If the distance from the one to the other is finite, then the projection is perspective: if the distance is infinite, the projection is parallel.

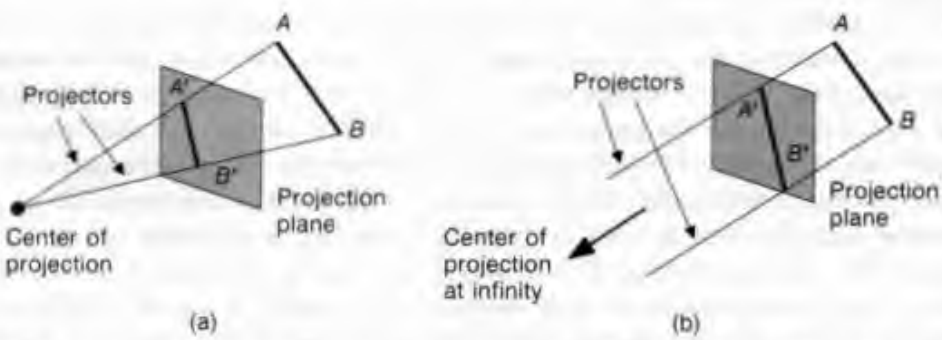


Fig. 6.2 (a) Line AB and its perspective projection $A'B'$. (b) Line AB and its parallel projection $A'B'$. Projectors AA' and BB' are parallel.

Samsung/Google, EX1027, p. 247

Figure 6.2 illustrates these two cases. The parallel projection is so named because, with the center of projection infinitely distant, the projectors are parallel. When defining a perspective projection, we explicitly specify its *center of projection*; for a parallel projection, we give its *direction of projection*. The center of projection, being a point, has homogeneous coordinates of the form $(x, y, z, 1)$. Since the direction of projection is a vector (i.e., a difference between points), it can be computed by subtracting two points $d = (x, y, z, 1) - (x', y', z', 1) = (a, b, c, 0)$. Thus, *directions* and *points at infinity* correspond in a natural way. A perspective projection whose center is a point at infinity becomes a parallel projection.

The visual effect of a perspective projection is similar to that of photographic systems and of the human visual system, and is known as *perspective foreshortening*: The size of the perspective projection of an object varies inversely with the distance of that object from the center of projection. Thus, although the perspective projection of objects tend to look realistic, it is not particularly useful for recording the exact shape and measurements of the objects; distances cannot be taken from the projection, angles are preserved only on those faces of the object parallel to the projection plane, and parallel lines do not in general project as parallel lines.

The parallel projection is a less realistic view because perspective foreshortening is lacking, although there can be different constant foreshortenings along each axis. The projection can be used for exact measurements and parallel lines do remain parallel. As with the perspective projection, angles are preserved only on faces of the object parallel to the projection plane.

The different types of perspective and parallel projections are discussed and illustrated at length in the comprehensive paper by Carlbom and Paciorek [CARL78]. In the following two subsections, we summarize the basic definitions and characteristics of the more commonly used projections; we then move on in Section 6.2 to understand how the projections are actually specified to PHIGS.

6.1.1 Perspective Projections

The perspective projections of any set of parallel lines that are not parallel to the projection plane converge to a *vanishing point*. In 3D, the parallel lines meet only at infinity, so the vanishing point can be thought of as the projection of a point at infinity. There is of course an infinity of vanishing points, one for each of the infinity of directions in which a line can be oriented.

If the set of lines is parallel to one of the three principal axes, the vanishing point is called an *axis vanishing point*. There are at most three such points, corresponding to the number of principal axes cut by the projection plane. For example, if the projection plane cuts only the z axis (and is therefore normal to it), only the z axis has a principal vanishing point, because lines parallel to either the y or x axes are also parallel to the projection plane and have no vanishing point.

Perspective projections are categorized by their number of principal vanishing points and therefore by the number of axes the projection plane cuts. Figure 6.3 shows two different one-point perspective projections of a cube. It is clear that they are one-point

Samsung/Google, EX1027, p. 248

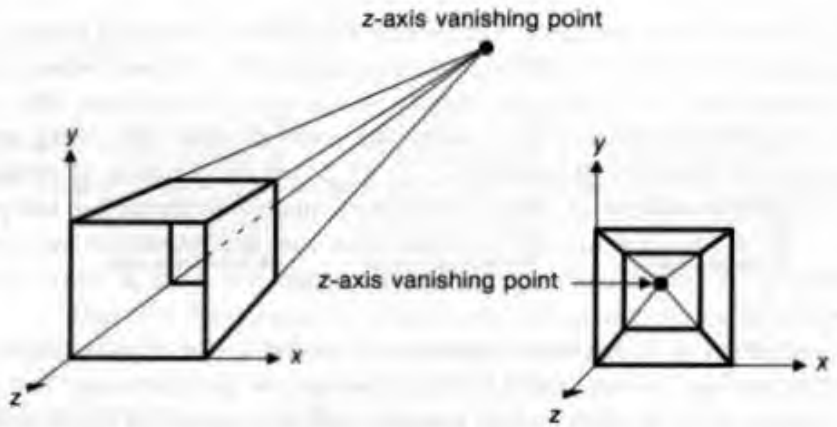


Fig. 6.3 One-point perspective projections of a cube onto a plane cutting the *z* axis, showing vanishing point of lines perpendicular to projection plane.

projections because lines parallel to the *x* and *y* axes do not converge; only lines parallel to the *z* axis do so. Figure 6.4 shows the construction of a one-point perspective with some of the projectors and with the projection plane cutting only the *z* axis.

Figure 6.5 shows the construction of a two-point perspective. Notice that lines parallel to the y axis do not converge in the projection. Two-point perspective is commonly used in architectural, engineering, industrial design, and in advertising drawings. Three-point perspectives are used less frequently, since they add little realism beyond that afforded by the two-point perspective.

6.1.2 Parallel Projections

Parallel projections are categorized into two types, depending on the relation between the direction of projection and the normal to the projection plane. In *orthographic* parallel

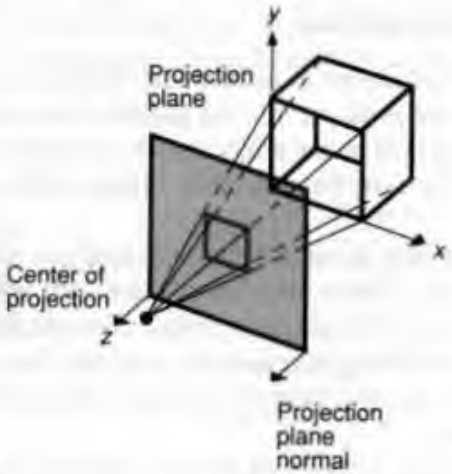


Fig. 6.4 Construction of one-point perspective projection of cube onto plane cutting the *z* axis. Projection-plane normal is parallel to *z* axis. (Adapted from [CARL78], Association for Computing Machinery, Inc.; used by permission.)

Samsung/Google, EX1027, p. 249

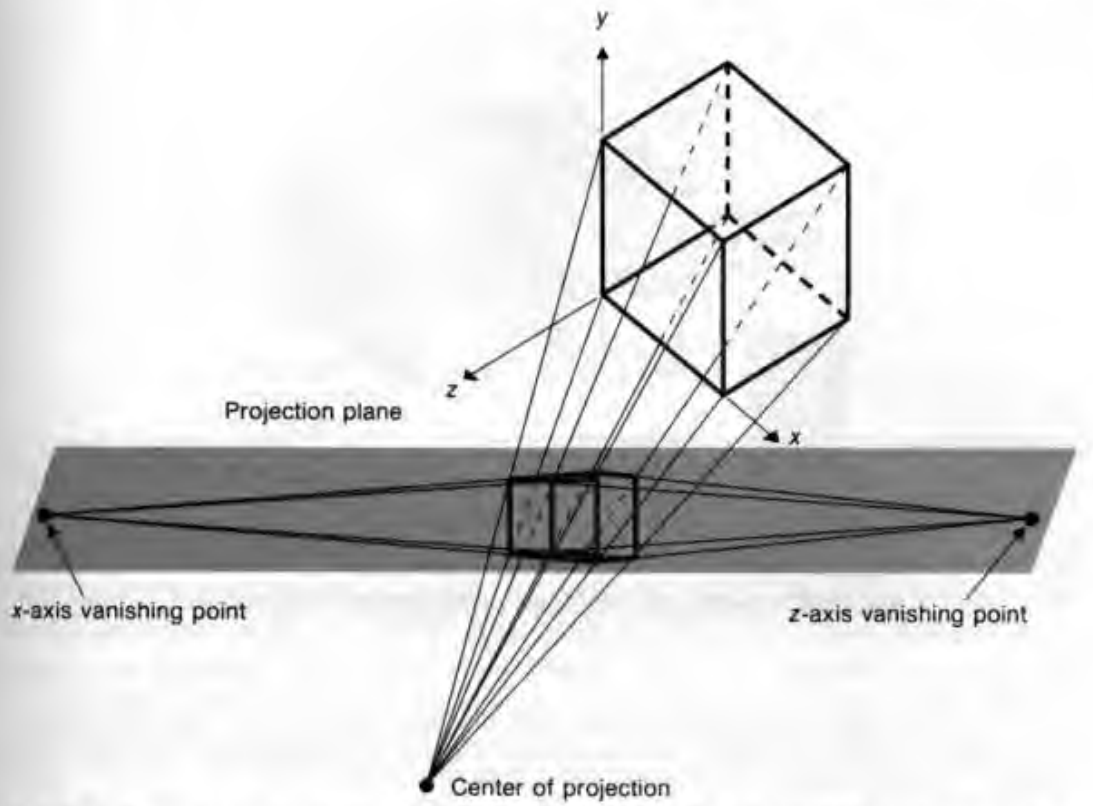


Fig. 6.5 Two-point perspective projection of a cube. The projection plane cuts the *x* and *z* axes.

projections, these directions are the same (or the reverse of each other), so the direction of projection is normal to the projection plane. For the *oblique* parallel projection, they are not.

The most common types of orthographic projections are the *front-elevation*, *top-elevation* (also called *plan-elevation*), and *side-elevation* projections. In all these, the projection plane is perpendicular to a principal axis, which is therefore the direction of projection. Figure 6.6 shows the construction of these three projections; they are often used in engineering drawings to depict machine parts, assemblies, and buildings, because distances and angles can be measured from them. Since each depicts only one face of an object, however, the 3D nature of the projected object can be difficult to deduce, even if several projections of the same object are studied simultaneously.

Axonometric orthographic projections use projection planes that are not normal to a principal axis and therefore show several faces of an object at once. They resemble the perspective projection in this way, but differ in that the foreshortening is uniform rather than being related to the distance from the center of projection. Parallelism of lines is preserved but angles are not, while distances can be measured along each principal axis (in general, with different scale factors).

The *isometric projection* is a commonly used axonometric projection. The projection-plane normal (and therefore the direction of projection) makes equal angles with each principal axis. If the projection-plane normal is (d_x, d_y, d_z), then we require that $|d_x| = |d_y|$

Samsung/Google, EX1027, p. 250

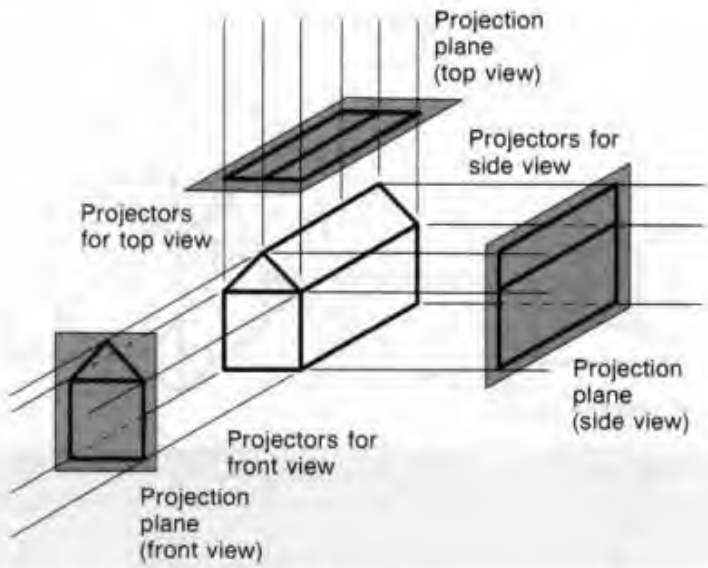


Fig. 6.6 Construction of three orthographic projections.

$= |d_z|$ or $\pm d_x = \pm d_y = \pm d_z$. There are just eight directions (one in each octant) that satisfy this condition. Figure 6.7 shows the construction of an isometric projection along one such direction, $(1, -1, -1)$.

The isometric projection has the useful property that all three principal axes are equally foreshortened, allowing measurements along the axes to be made to the same scale (hence the name: *iso* for equal, *metric* for measure). In addition, the principal axes project so as to make equal angles one with another, as shown in Fig. 6.8.

Oblique projections, the second class of parallel projections, differ from orthographic projections in that the projection-plane normal and the direction of projection differ.

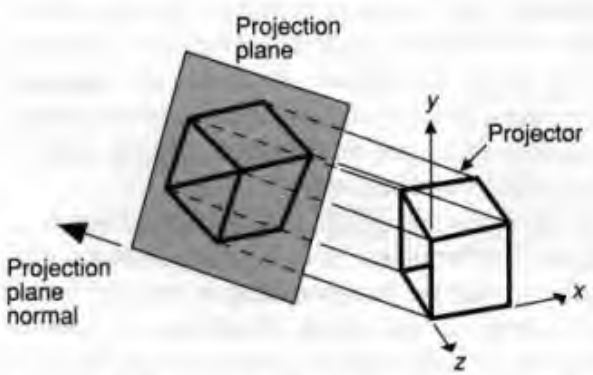


Fig. 6.7 Construction of an isometric projection of a unit cube. (Adapted from [CARL78], Association for Computing Machinery, Inc.; used by permission.)

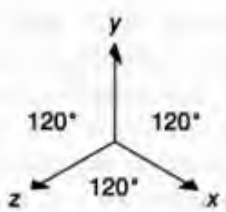


Fig. 6.8 Isometric projection of unit vectors, with direction of projection (1, 1, 1).

Samsung/Google, EX1027, p. 251

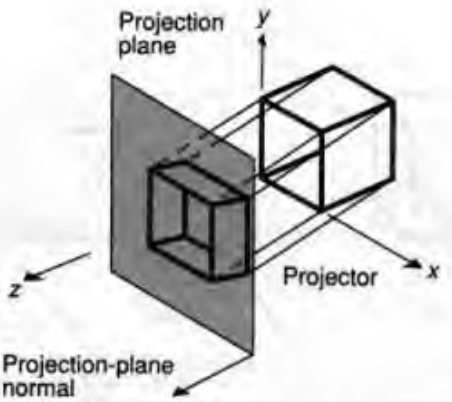


Fig. 6.9 Construction of oblique projection. (Adapted from [CARL78], Association for Computing Machinery, Inc.; used by permission.)

Oblique projections combine properties of the front, top, and side orthographic projections with those of the axonometric projection: the projection plane is normal to a principal axis, so the projection of the face of the object parallel to this plane allows measurement of angles and distances. Other faces of the object project also, allowing distances along principal axes, but not angles, to be measured. Oblique projections are widely, although not exclusively, used in this text because of these properties and because they are easy to draw. Figure 6.9 shows the construction of an oblique projection. Notice that the projection-plane normal and the direction of projection are not the same.

Two frequently used oblique projections are the *cavalier* and the *cabinet*. For the cavalier projection, the direction of projection makes a 45° angle with the projection plane. As a result, the projection of a line perpendicular to the projection plane has the same length as the line itself; that is, there is no foreshortening. Figure 6.10 shows several cavalier projections of the unit cube onto the (x, y) plane; the receding lines are the projections of the cube edges that are perpendicular to the (x, y) plane, and they form an angle α to the horizontal. This angle is typically 30° or 45°.

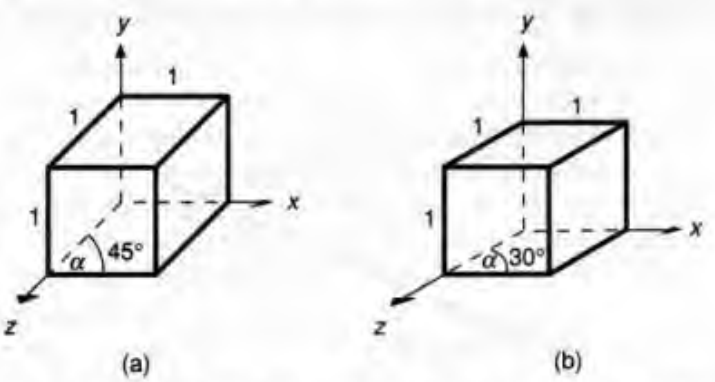


Fig. 6.10 Cavalier projection of the unit cube onto the $z = 0$ plane. All edges project at unit length. In (a), the direction of projection is $(\sqrt{2}/2, \sqrt{2}/2, -1)$; in (b), it is $(\sqrt{3}/2, 1/2, -1)$.

Samsung/Google, EX1027, p. 252

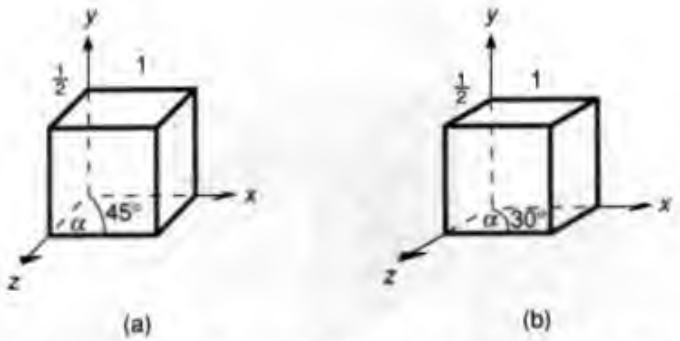


Fig. 6.11 Cabinet projection of the unit cube onto the $z = 0$ plane. Edges parallel to the x and y axes project at unit length. In (a), the direction of projection is $(\sqrt{2}/4, \sqrt{2}/4, -1)$; in (b), it is $(\sqrt{3}/4, 1/4, -1)$.

Cabinet projections, such as those in Fig. 6.11, have a direction of projection that makes an angle of $\arctan(2) = 63.4°$ with the projection plane, so lines perpendicular to the projection plane project at one-half their actual length. Cabinet projections are a bit more realistic than cavalier ones are, since the foreshortening by one-half is more in keeping with our other visual experiences.

Figure 6.12 helps to explain the angles made by projectors with the projection plane for cabinet and cavalier projections. The (x, y) plane is the projection plane and the point P' is the projection of $(0, 0, 1)$ onto the projection plane. The angle α and length l are the same as are used in Figs. 6.10 and 6.11, and we can control them by varying the direction of projection (l is the length at which the z-axis unit vector projects onto the (x, y) plane; α is the angle the projection makes with the x axis). Designating the direction of projection as $(dx, dy, -1)$, we see from Fig. 6.12 that $dx = l\cos\alpha$ and $dy = l\sin\alpha$. Given a desired l and α, the direction of projection is $(l\cos\alpha, l\sin\alpha, -1)$.

Figure 6.13 shows the logical relationships among the various types of projections. The common thread uniting them all is that they involve a projection plane and either a center of projection for the perspective projection, or a direction of projection for the parallel projection. We can unify the parallel and perspective cases further by thinking of the center of projection as defined by the direction to the center of projection from some reference

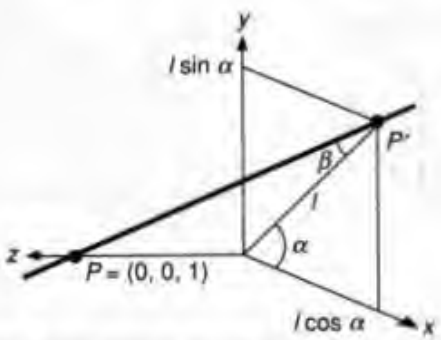


Fig. 6.12 Oblique parallel projection of $P = (0, 0, 1)$ onto $P' = (l\cos\alpha, l\sin\beta, 0)$. The direction of projection is $P' - P = (l\cos\alpha, l\sin\beta, -1)$.

Samsung/Google, EX1027, p. 253

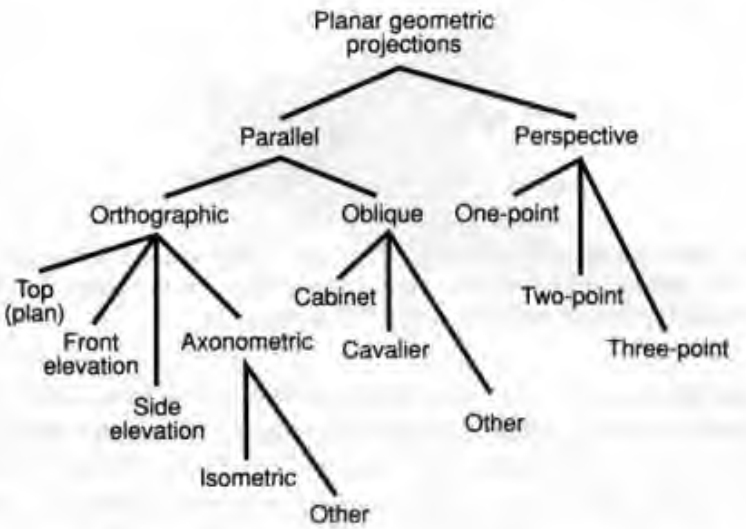


Fig. 6.13 The subclasses of planar geometric projections. *Plan view* is another term for a top view. *Front* and *side* are often used without the term *elevation*.

point, plus the distance to the reference point. When this distance increases to infinity, the projection becomes a parallel projection. Hence, we can also say that the common thread uniting these projections is that they involve a projection plane, a direction to the center of projection, and a distance to the center of projection.

In the next section, we consider how to integrate these various types of projections into the 3D viewing process.

6.2 SPECIFYING AN ARBITRARY 3D VIEW

As suggested by Fig. 6.1, 3D viewing involves not just a projection but also a view volume against which the 3D world is clipped. The projection and view volume together provide all the information needed to clip and project into 2D space. Then, the 2D transformation into physical device coordinates is straightforward. We now build on the planar-geometric-projection concepts introduced in the preceding section to show how to specify a view volume. The viewing approach and terminology presented here is that used in PHIGS.

The projection plane, henceforth called the *view plane* to be consistent with the graphics literature, is defined by a point on the plane called the *view reference point* (VRP) and a normal to the plane called the *view-plane normal* (VPN).[1] The view plane may be anywhere with respect to the world objects to be projected: it may be in front of, cut through, or be behind the objects.

Given the view plane, a window on the view plane is needed. The window's role is similar to that of a 2D window: its contents are mapped into the viewport, and any part of

[1]PHIGS has an additional variable, the view-plane distance (VPD): the view plane can be a distance VPD from the VRP. VPD is positive in the direction of VPN. See Exercise 6.22.

Samsung/Google, EX1027, p. 254

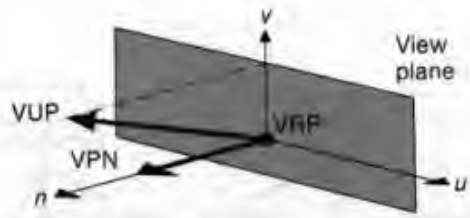


Fig. 6.14 The view plane is defined by VPN and VRP; the *v* axis is defined by the projection of VUP along VPN onto the view plane. The *u* axis forms the right-handed viewing reference-coordinate system with VPN and *v*.

the 3D world that projects onto the view plane outside of the window is not displayed. We shall see that the window also plays an important role in defining the view volume.

To define a window on the view plane, we need some means of specifying minimum and maximum window coordinates along two orthogonal axes. These axes are part of the 3D *viewing-reference coordinate* (VRC) system. The origin of the VRC system is the VRP. One axis of the VRC is VPN; this axis is called the *n* axis. A second axis of the VRC is found from the *view up vector* (VUP), which determines the *v*-axis direction on the view plane. The *v* axis is defined such that the projection of VUP parallel to VPN onto the view plane is coincident with the *v* axis. The *u*-axis direction is defined such that *u*, *v*, and *n* form a right-handed coordinate system, as in Fig. 6.14. The VRP and the two direction vectors VPN and VUP are specified in the right-handed world-coordinate system. (Some graphics packages use the *y* axis as VUP, but this is too restrictive and fails if VPN is parallel to the *y* axis, in which case VUP would be undefined.)

With the VRC system defined, the window's minimum and maximum *u* and *v* values can be defined, as in Fig. 6.15. This figure illustrates that the window need not be symmetrical about the view reference point, and explicitly shows the center of the window, CW.

The center of projection and direction of projection (DOP) are defined by a *projection reference point* (PRP) plus an indicator of the projection type. If the projection type is perspective, then PRP is the center of projection. If the projection type is parallel, then the DOP is from the PRP to CW. The CW is in general *not* the VRP, which need not even be within the window bounds.

The PRP is specified in the VRC system, not in the world-coordinate system; thus, the position of the PRP relative to the VRP does not change as VUP or VRP are moved. The

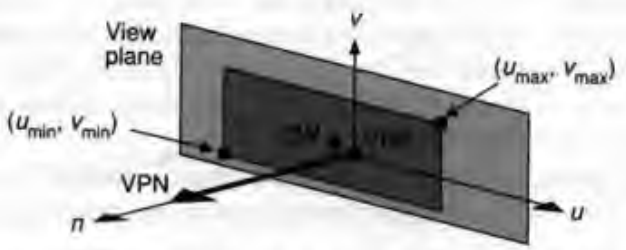


Fig. 6.15 The view reference-coordinate system (VRC) is a right-handed system made up of the *u*, *v*, and *n* axes. The *n* axis is always the VPN. CW is the center of the window.

Samsung/Google, EX1027, p. 255

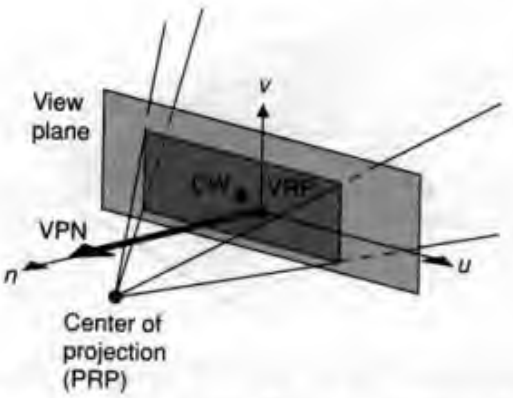


Fig. 6.16 The semi-infinite pyramid view volume for perspective projection. CW is the center of the window.

advantage of this is that the programmer can specify the direction of projection required, for example, by a cavalier projection, and then change VPN and VUP (hence changing VRC), without having to recalculate the PRP needed to maintain the desired projection. On the other hand, moving the PRP about to get different views of an object may be more difficult.

The *view volume* bounds that portion of the world that is to be clipped out and projected onto the view plane. For a perspective projection, the view volume is the semi-infinite pyramid with apex at the PRP and edges passing through the corners of the window. Figure 6.16 shows a perspective-projection view volume. Positions behind the center of projection are not included in the view volume and thus are not projected. In reality, of course, our eyes see an irregularly shaped conelike view volume. However, a pyramidal view volume is mathematically more tractable, and is consistent with the concept of a rectangular viewport.

For parallel projections, the view volume is an infinite parallelepiped with sides parallel to the direction of projection, which is the direction from the PRP to the center of the window. Figures 6.17 and 6.18 show parallel-projection view volumes and their

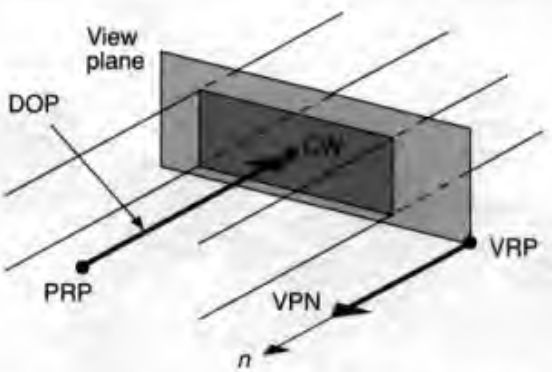


Fig. 6.17 Infinite parallelepiped view volume of parallel orthographic projection. The VPN and direction of projection (DOP) are parallel. DOP is the vector from PRP to CW, and is parallel to the VPN.

Samsung/Google, EX1027, p. 256

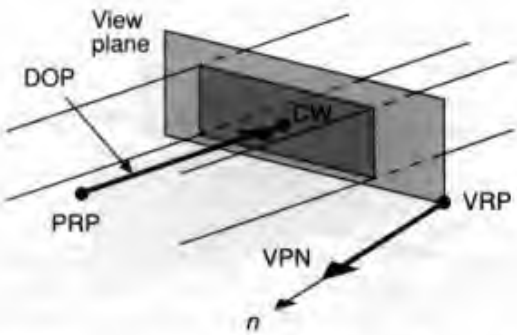


Fig. 6.18 Infinite parallelepiped view volume of oblique orthographic projection. The direction of projection (DOP) is not parallel to the VPN.

relation to the view plane, window, and PRP. For orthographic parallel projections, but not for oblique parallel projections, the sides of the view volume are normal to the view plane.

At times, we might want the view volume to be finite, in order to limit the number of output primitives projected onto the view plane. Figures 6.19, 6.20, and 6.21 show how the view volume is made finite with a *front clipping plane* and *back clipping plane*.These planes, sometimes called the *hither* and *yon* planes, are parallel to the view plane; their normal is the VPN. The planes are specified by the signed quantities *front distance* (F) and *back distance* (B) relative to the view reference point and along the VPN, with positive distances in the direction of the VPN. For the view volume to be positive, the front distance must be algebraically greater than the back distance.

Limiting the view volume in this way can be useful in order to eliminate extraneous objects and to allow the user to concentrate on a particular portion of the world. Dynamic modification of either the front or rear distances can give the viewer a good sense of the spatial relationships between different parts of the object as these parts appear and disappear from view (see Chapter 14). For perspective projections, there is an additional motivation. An object very distant from the center of projection projects onto the view surface as a

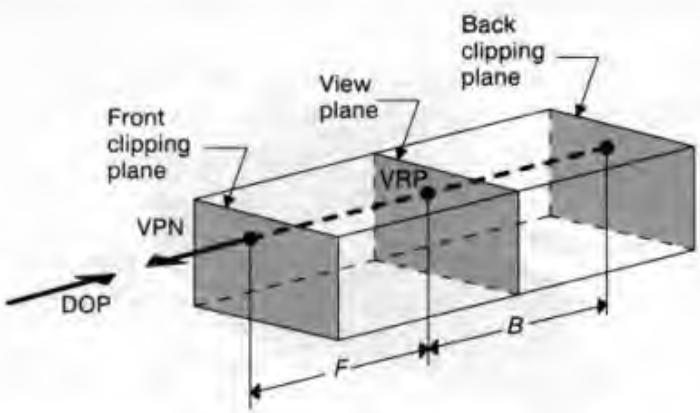


Fig. 6.19 Truncated view volume for an orthographic parallel projection. DOP is the direction of projection.

Samsung/Google, EX1027, p. 257

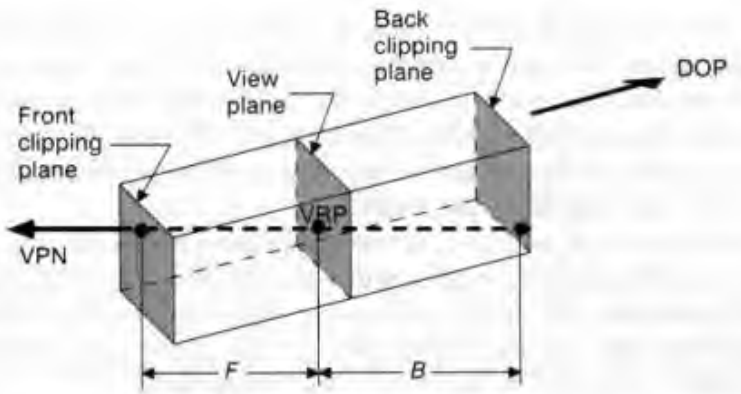


Fig. 6.20 Truncated view volume for oblique parallel projection showing VPN oblique to direction of projection (DOP); VPN is also normal to the front and back clipping planes.

"blob" of no distinguishable form. In displaying such an object on a plotter, the pen can wear through the paper; on a vector display, the CRT phosphor can be burned by the electron beam; and on a film recorder, the high concentration of light causes a fuzzy white area to appear. Also, an object very near the center of projection may extend across the window like so many disconnected pick-up sticks, with no discernible structure. Specifying the view volume appropriately can eliminate such problems.

How are the contents of the view volume mapped onto the display surface? First, consider the unit cube extending from 0 to 1 in each of the three dimensions of *normalized projection coordinates* (NPC). The view volume is transformed into the rectangular solid of NPC, which extends from x_{min} to x_{max} along the x axis, from y_{min} to y_{max} along the y axis, and from z_{min} to z_{max} along the z axis. The front clipping plane becomes the z_{max} plane and the back clipping plane becomes the z_{min} plane. Similarly, the u_{min} side of the view volume becomes the x_{min} plane and the u_{max} side becomes the x_{max} plane. Finally, the v_{min} side of the view volume becomes the y_{min} plane and the v_{max} side becomes the y_{max} plane. This rectangular solid portion of NPC, called a *3D viewport*, is within the unit cube.

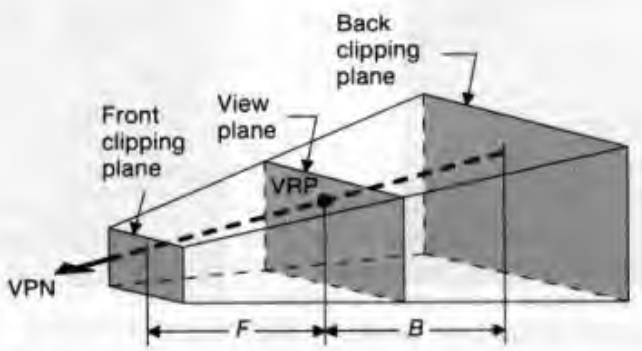


Fig. 6.21 Truncated view volume for perspective projection.

Samsung/Google, EX1027, p. 258

The $z = 1$ face of this unit cube, in turn, is mapped into the largest square that can be inscribed on the display. To create a wire-frame display of the contents of the 3D viewport (which are the contents of the view volume), the z-component of each output primitive is simply discarded, and the output primitive is displayed. We will see in Chapter 15 that hidden surface removal simply uses the z-component to determine which output primitives are closest to the viewer and hence are visible.

PHIGS uses two 4×4 matrices, the view orientation matrix and the view mapping matrix, to represent the complete set of viewing specifications. The VRP, VPN, and VUP are combined to form the *view orientation matrix*, which transforms positions represented in world coordinates into positions represented in VRC. This is the transformation that takes the u, v, and n axes into the x, y, and z axes, respectively.

The view volume specifications, given by PRP, u_{min}, u_{max}, v_{min}, v_{max}, F, and B, along with the 3D viewport specification, given by x_{min}, x_{max}, y_{min}, y_{max}, z_{min}, z_{max}, are combined to form the *view mapping matrix*, which transforms points in VRC to points in normalized projection coordinates. The subroutine calls that form the view orientation matrix and view mapping matrix are discussed in Section 7.3.4.

In the next section, we see how to obtain various views using the concepts introduced in this section. In Section 6.4 the basic mathematics of planar geometric projections is introduced, whereas in Section 6.5 the mathematics and algorithms needed for the entire viewing operation are developed.

6.3 EXAMPLES OF 3D VIEWING

In this section, we consider how the basic viewing concepts introduced in the previous section can be applied to create a variety of projections, such as those shown in Figs. 6.22 and 6.23. Because the house shown in these figures is used throughout this section, it will be helpful to remember its dimensions and position, which are indicated in Fig. 6.24. For each view discussed, we give a table showing the VRP, VPN, VUP, PRP, window, and



Fig. 6.22 Two-point perspective projection of a house.

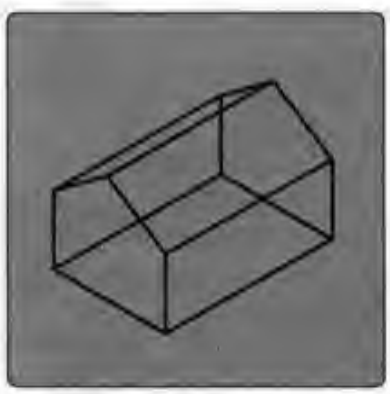


Fig. 6.23 Isometric projection of a house.

Samsung/Google, EX1027, p. 259

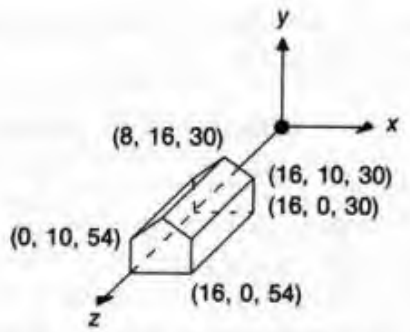


Fig. 6.24 The house extends from 30 to 54 in *z*, from 0 to 16 in *x*, and from 0 to 16 in *y*.

projection type (perspective or parallel). The 3D viewport default, which is the unit cube in NPC, is assumed throughout this section. The notation (WC) or (VRC) is added to the table as a reminder of the coordinate system in which the viewing parameter is given. The form of the table is illustrated here for the default viewing specification used by PHIGS. The defaults are shown in Fig. 6.25(a). The view volume corresponding to these defaults is

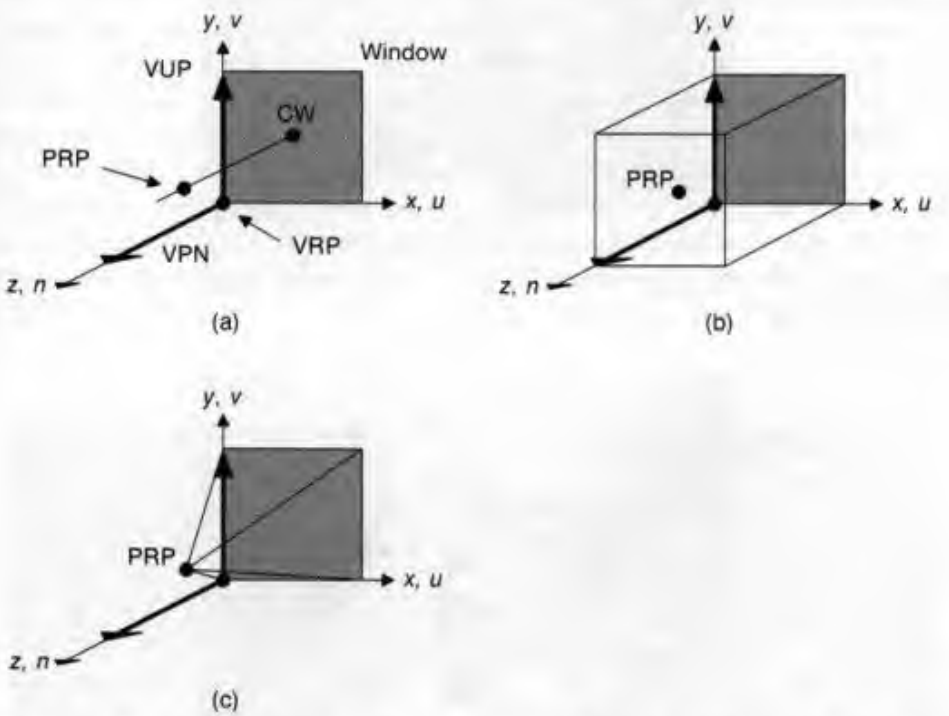


Fig. 6.25 (a) The default viewing specification: VRP is at the origin, VUP is the *y* axis, and VPN is the *z* axis. This makes the VRC system of *u*, *v*, and *n* coincide with the *x*, *y*, *z* world-coordinate system. The window extends from 0 to 1 along *u* and *v*, and PRP is at (0.5, 0.5, 1.0). (b) Default parallel projection view volume. (c) View volume if default projection were perspective.

Samsung/Google, EX1027, p. 260

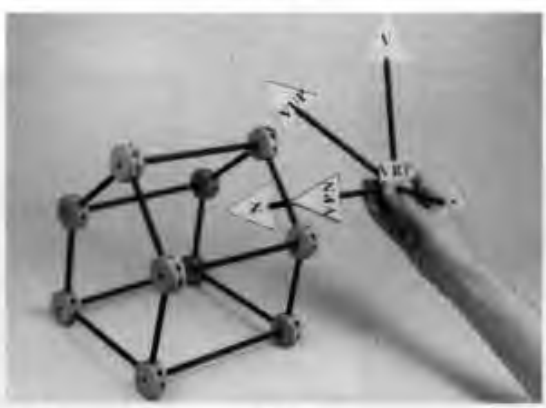
shown in Fig. 6.25(b). If the type of projection is perspective rather than parallel, then the view volume is the pyramid shown in Fig. 6.25(c).

Viewing parameter	Value	Comments
VRP(WC)	(0, 0, 0)	origin
VPN(WC)	(0, 0, 1)	z axis
VUP(WC)	(0, 1, 0)	y axis
PRP(VRC)	(0.5, 0.5, 1.0)	
window (VRC)	(0, 1, 0, 1)	
projection type	parallel	

Readers wanting to review how all these parameters interrelate are encouraged to construct a house, the world coordinate system, and the VRC system with Tinker Toys, as pictured in Fig. 6.26. The idea is to position the VRC system in world coordinates as in the viewing example and to imagine projectors from points on the house intersecting the view plane. In our experience, this is a useful way to understand (and teach) 3D viewing concepts.



(a)



(b)

Fig. 6.26 Stick models useful for understanding 3D viewing. (a) House and world-coordinate system. (b) House and VRC system.

Samsung/Google, EX1027, p. 261

6.3.1 Perspective Projections

To obtain the front one-point perspective view of the house shown in Fig. 6.27 (this and all similar figures were made with the SPHIGS program, discussed in Chapter 7), we position the center of projection (which can be thought of as the position of the viewer) at $x = 8$, $y = 6$, and $z = 84$. The x value is selected to be at the horizontal center of the house and the y value to correspond to the approximate eye level of a viewer standing on the (x, z) plane; the z value is arbitrary. In this case, z is removed 30 units from the front of the house ($z = 54$ plane). The window has been made quite large, to guarantee that the house fits within the view volume. All other viewing parameters have their default values, so the overall set of viewing parameters is as follows:

VRP(WC)	(0, 0, 0)
VPN(WC)	(0, 0, 1)
VUP(WC)	(0, 1, 0)
PRP(VRC)	(8, 6, 84)
window(VRC)	(−50, 50, −50, 50)
projection type	perspective

Although the image in Fig. 6.27 is indeed a perspective projection of the house, it is very small and is not centered on the view surface. We would prefer a more centered projection of the house that more nearly spans the entire view surface, as in Fig. 6.28. We can produce this effect more easily if the view plane and the front plane of the house coincide. Now, because the front of the house extends from 0 to 16 in both x and y, a window extending from −1 to 17 in x and y produces reasonable results.

We place the view plane on the front face of the house by placing the VRP anywhere in the $z = 54$ plane; (0, 0, 54), the lower-left front corner of the house, is fine. For the center of projection to be the same as in Fig. 6.27, the PRP, which is in the VRC system, needs to be at (8, 6, 30). Figure 6.29 shows this new arrangement of the VRC, VRP, and PRP,

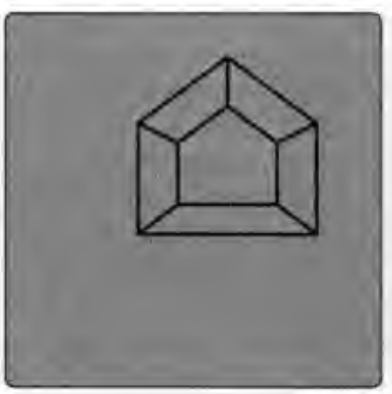


Fig. 6.27 One-point perspective projection of the house.

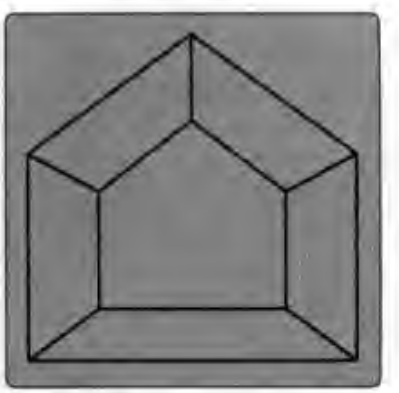


Fig. 6.28 Centered perspective projection of a house.

Samsung/Google, EX1027, p. 262

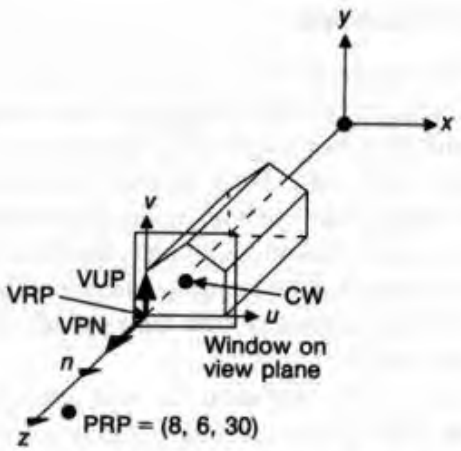


Fig. 6.29 The viewing situation for Fig. 6.28.

which corresponds to the following set of viewing parameters:

VRP(WC)	(0, 0, 54)
VPN(WC)	(0, 0, 1)
VUP(WC)	(0, 1, 0)
PRP(VRC)	(8, 6, 30)
window(VRC)	(−1, 17, −1, 17)
projection type	perspective

This same result can be obtained in many other ways. For instance, with the VRP at (8, 6, 54), as in Fig. 6.30, the center of projection, given by the PRP, becomes (0, 0, 30). The window must also be changed, because its definition is based on the VRC system, the origin of which is the VRP. The appropriate window extends from −9 to 9 in u and from −7 to 11 in v. With respect to the house, this is the same window as that used in the previous

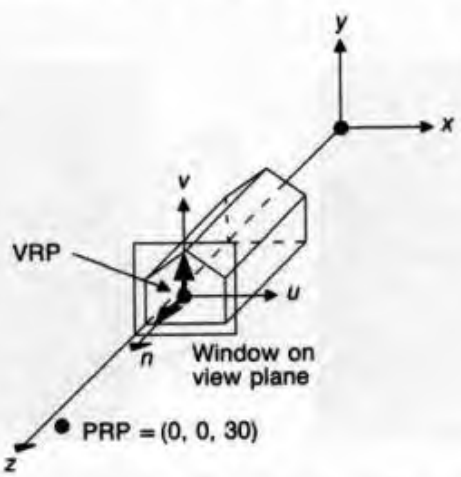


Fig. 6.30 An alternative viewing situation for Fig. 6.28.

Samsung/Google, EX1027, p. 263

example, but it is now specified in a different VRC system. Because the view-up direction is the y axis, the u axis and x axis are parallel, as are the v and y axes. In summary, the following viewing parameters, shown in Fig. 6.30, also produce Fig. 6.28:

VRP(WC)	(8, 6, 54)
VPN(WC)	(0, 0, 1)
VUP(WC)	(0, 1, 0)
PRP(VRC)	(0, 0, 30)
window(VRC)	(−9, 9, −7, 11)
projection type	perspective

Next, let us try to obtain the two-point perspective projection shown in Fig. 6.22. The center of projection is analogous to the position of a camera that takes snapshots of world-coordinate objects. With this analogy in mind, the center of projection in Fig. 6.22 seems to be somewhat above and to the right of the house, as viewed from the positive z axis. The exact center of projection is (36, 25, 74). Now, if the corner of the house at (16, 0, 54) is chosen as the VRP, then this center of projection is at (20, 25, 20) relative to it. With the view plane coincident with the front of the house (the $z = 54$ plane), a window ranging from −20 to 20 in u and from −5 to 35 in v is certainly large enough to contain the projection. Hence, we can specify the view of Fig. 6.31 with the viewing parameters:

VRP(WC)	(16, 0, 54)
VPN(WC)	(0, 0, 1)
VUP(WC)	(0, 1, 0)
PRP(VRC)	(20, 25, 20)
window(VRC)	(−20, 20, −5, 35)
projection type	perspective

This view is similar to, but clearly is not the same as, that in Fig. 6.22. For one thing, Fig. 6.22 is a two-point perspective projection, whereas Fig. 6.31 is a one-point perspective. It is apparent that simply moving the center of projection is not sufficient to produce Fig. 6.22. In fact, we need to reorient the view plane such that it cuts both the x and

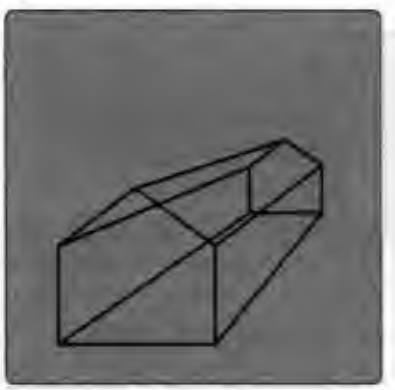


Fig. 6.31 Perspective projection of a house from (36, 25, 74) with VPN parallel to the z axis.

Samsung/Google, EX1027, p. 264

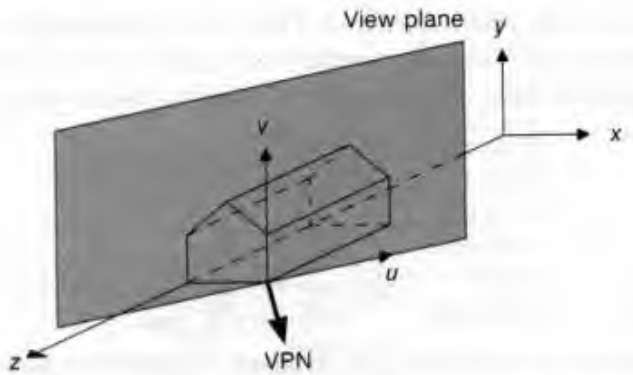


Fig. 6.32 The view plane and VRC system corresponding to Fig. 6.22.

z axes, by setting VPN to (1, 0, 1). Thus, the viewing parameters for Fig. 6.22 are as follows:

VRP(WC)	(16, 0, 54)
VPN(WC)	(1, 0, 1)
VUP(WC)	(0, 1, 0)
PRP(VRC)	$(0, 25, 20\sqrt{2})$
window(VRC)	(−20, 20, −5, 35)
projection type	perspective

Figure 6.32 shows the view plane established with this *VPN*. The $20\sqrt{2}$ n component of the *PRP* is used so that the center of projection is a distance $20\sqrt{2}$ away from the VRP in the (x, y) plane, as shown in Fig. 6.33.

There are two ways to choose a window that completely surrounds the projection, as does the window in Fig. 6.22. One can estimate the size of the projection of the house onto the view plane using a sketch, such as Fig. 6.33, to calculate the intersections of projectors

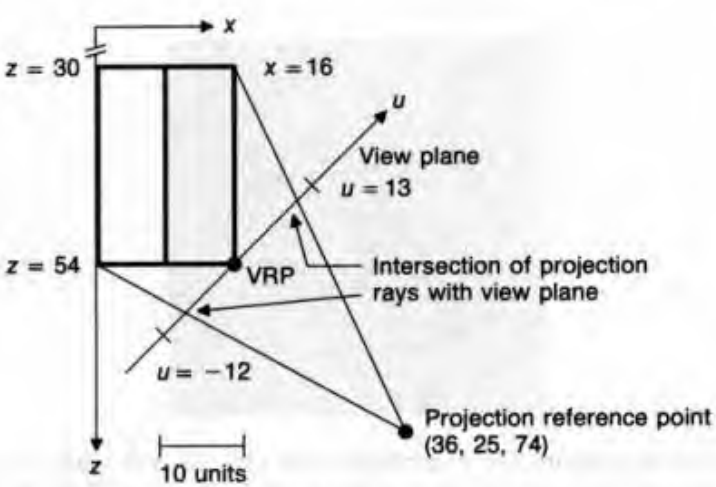


Fig. 6.33 Top (plan) view of a house for determining an appropriate window size.

Samsung/Google, EX1027, p. 265

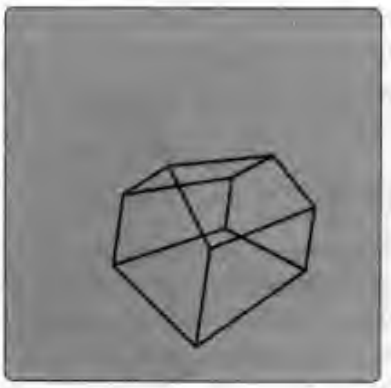


Fig. 6.34 Projection of house produced by rotating VUP.

with the view plane. A better alternative, however, is to allow the window bounds to be variables in a program that are determined interactively via a valuator or locator device.

Figure 6.34 is obtained from the same projection as is Fig. 6.22, but the window has a different orientation. In all previous examples, the v axis of the VRC system was parallel to the y axis of the world-coordinate system; thus, the window (two of whose sides are parallel to the v axis) was nicely aligned with the vertical sides of the house. Figure 6.34 has exactly the same viewing parameters as does Fig. 6.22, except that VUP has been rotated away from the y axis by about 10°.

Another way to specify viewing parameters for perspective projections is suggested in Fig. 6.35. This figure is modeled after the way a photographer might think about positioning a camera. Six parameters are needed: the center of projection, which is the camera position; the center of attention, which is a point at which the camera is aimed (the VPN is the vector from the center of attention to the center of projection); VUP, the up vector; D, the distance from the center of projection to the projection plane; W, the width of the window on the projection plane, and H, the height of the window on the projection plane. The center of attention need not be on the view plane. In this model, the VPN is

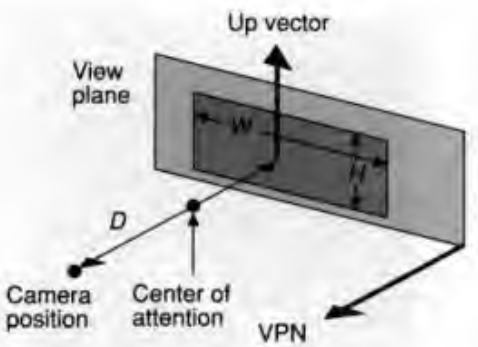


Fig. 6.35 Another way to specify a view, with a camera position (the center of projection), center of attention, up vector, distance from the center of projection to the projection plane, and the height and width of the window on the projection plane. VPN is parallel to the direction from the center of attention to the camera position.

Samsung/Google, EX1027, p. 266

always pointed directly at the center of projection, and the view volume is symmetrical about its center line. The positions are all given in world coordinates—there is no concept of viewing coordinates. Exercise 6.24 asks you to convert from these six viewing parameters into the viewing model given here.

6.3.3 Parallel Projections

We create a front parallel projection of the house (Fig. 6.36) by making the direction of projection parallel to the z axis. Recall that the direction of projection is determined by the PRP and by the center of the window. With the default VRC system and a window of $(-1, 17, -1, 17)$, the center of the window is $(8, 8, 0)$. A PRP of $(8, 8, 100)$ provides a direction of projection parallel to the z axis. Figure 6.37 shows the viewing situation that creates Fig. 6.36. The viewing parameters are as follows:

VRP(WC)	(0, 0, 0)
VPN(WC)	(0, 0, 1)
VUP(WC)	(0, 1, 0)
PRP(VRC)	(8, 8, 100)
window(VRC)	(−1, 17, −1, 17)
projection type	parallel

To create the side view (Fig. 6.38), we require the viewing situation of Fig. 6.39, with the (y, z) plane (or any plane parallel to it) as the view plane. This situation corresponds to the following viewing parameters:

VRP(WC)	(0, 0, 54)	
VPN(WC)	(1, 0, 0)	x axis
VUP(WC)	(0, 1, 0)	y axis
PRP(VRC)	(12, 8, 16)	
window(VRC)	(−1, 25, −5, 21)	
projection type	parallel	



Fig. 6.36 Front parallel projection of the house.

Samsung/Google, EX1027, p. 267

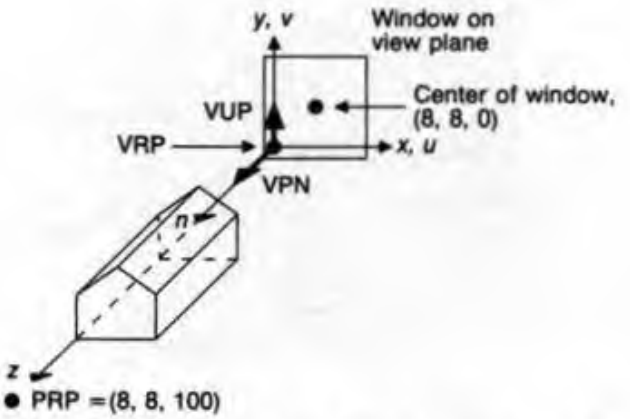


Fig. 6.37 Viewing situation that creates Fig. 6.36, a front view of the house. The PRP could be any point with $x = 8$ and $y = 8$.

The center of the window is at (12, 8, 0) in VRC; hence, the PRP has these same u and v coordinates.

We create a top view of the house by using the (x, z) plane as the view plane and VPN as the y axis. The default view-up direction of $+y$ must be changed; we use the negative x axis. With VRP again specified as a corner of the house, we have the viewing situation in Fig. 6.40, defined by the following viewing parameters:

VRP(WC)	(16, 0, 54)	
VPN(WC)	(0, 1, 0)	y axis
VUP(WC)	(−1, 0, 0)	negative x axis
PRP(VRC)	(12, 8, 30)	
window(VRC)	(−1, 25, −5, 21)	
projection type	parallel	



Fig. 6.38 Parallel projection from the side of the house.

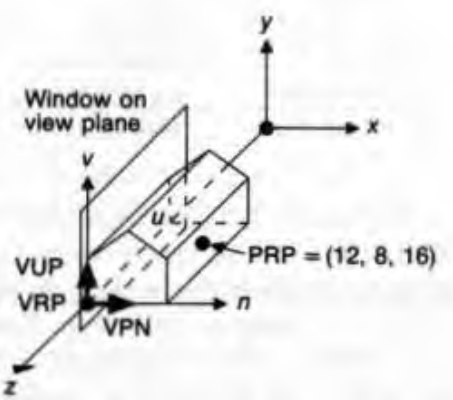


Fig. 6.39 The viewing situation for Fig. 6.38.

Samsung/Google, EX1027, p. 268

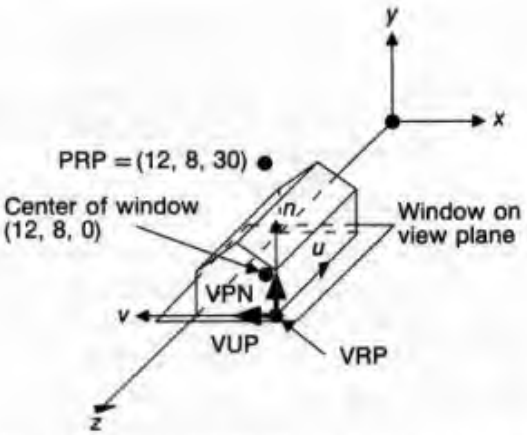


Fig. 6.40 The viewing situation for a top view of the house.

Figure 6.23 is an isometric (parallel orthographic) projection in the direction (−1, −1, −1), one of the eight possible directions for an isometric (see Section 6.1.2). The following viewing parameters create such an isometric projection:

VRP(WC)	(8, 8, 42)
VPN(WC)	(1, 1, 1)
VUP(WC)	(0, 1, 0)
PRP(VRC)	(0, 0, 10)
window(VRC)	(−20, 20, −20, 20)
projection type	parallel

A cavalier projection with angle α (see Fig. 6.10) is specified with

VRP(WC)	(8, 8, 54)	middle front of house
VPN(WC)	(0, 0, 1)	z axis
VUP(WC)	(0, 1, 0)	y axis
PRP(VRC)	($\cos\alpha$, $\sin\alpha$, 1)	
window(VRC)	(−15, 15, −15, 15)	
projection type	parallel	

The window is symmetrical about VRP, which implies that VRP, the origin of the VRC system, is the window's center. Placing PRP as specified means that the direction of projection, which is the vector from PRP to the window center, is (0, 0, 0) − PRP = − PRP, or its negative PRP. From Section 6.1.2, we know that the cavalier projection's direction of projection is that given in the preceding table: − PRP = − ($\cos\alpha$, $\sin\alpha$, 1). Now notice what happens if a cavalier projection onto a plane of constant x instead of constant z is desired: only VRP, VPN, and VUP need to be changed to establish a new VRC system. The PRP, window, and projection type can remain fixed.

Samsung/Google, EX1027, p. 269

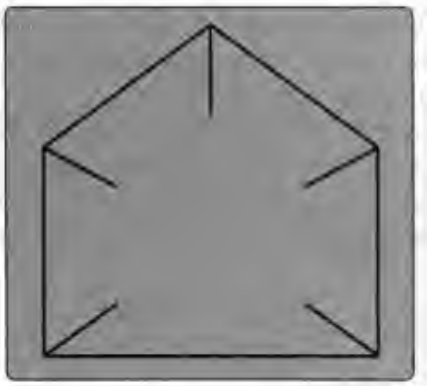


Fig. 6.41 Perspective projection of the house with back clipping plane at $z = 31$.

6.3.4 Finite View Volumes

In all the examples so far, the view volume has been assumed to be infinite. The front and back clipping planes, described Section 6.2, help to determine a *finite view volume*. These planes, both of which are parallel to the view plane, are at distances F and B respectively from the view reference point, measured from VRP along VPN. To avoid a negative view volume, we must ensure that F is algebraically greater than B.

A front perspective view of the house with the rear wall clipped away (Fig. 6.41) results from the following viewing specification, in which F and B have been added. If a distance is given, then clipping against the corresponding plane is assumed; otherwise, it is not. The viewing specification is as follows:

VRP(WC)	(0, 0, 54)	lower-left front of house
VPN(WC)	(0, 0, 1)	z axis
VUP(WC)	(0, 1, 0)	y axis
PRP(VRC)	(8, 6, 30)	
window(VRC)	(−1, 17, −1, 17)	
projection type	perspective	
F(VRC)	+1	one unit in front of house, at $z = 54 + 1 = 55$
B(VRC)	−23	one unit from back of house, at $z = 54 - 23 = 31$

The viewing situation for this case is the same as that in Fig. 6.29, except for the addition of the clipping planes.

If the front and back clipping planes are moved dynamically, the 3D structure of the object being viewed can often be discerned more readily than it can with a static view.

6.4 THE MATHEMATICS OF PLANAR GEOMETRIC PROJECTIONS

In this section, we introduce the basic mathematics of planar geometric projections. For simplicity, we start by assuming that, in the perspective projection, the projection plane is normal to the z axis at $z = d$, and that, in the parallel projection, the projection plane is the $z = 0$ plane. Each of the projections can be defined by a 4×4 matrix. This is convenient,

Samsung/Google, EX1027, p. 270

because the projection matrix can be composed with transformation matrices, allowing two operations (transform, then project) to be represented as a single matrix. In the next section, we discuss arbitrary projection planes.

In this section, we derive 4 × 4 matrices for several projections, beginning with a projection plane at a distance d from the origin and a point P to be projected onto it. To calculate $P_p = (x_p, y_p, z_p)$, the perspective projection of (x, y, z) onto the projection plane at $z = d$, we use the similar triangles in Fig. 6.42 to write the ratios

$$\frac{x_p}{d} = \frac{x}{z}; \qquad \frac{y_p}{d} = \frac{y}{z}. \tag{6.1}$$

Multiplying each side by d yields

$$x_p = \frac{d \cdot x}{z} = \frac{x}{z/d}, \qquad y_p = \frac{d \cdot y}{z} = \frac{y}{z/d}. \tag{6.2}$$

The distance d is just a scale factor applied to x_p and y_p. The division by z causes the perspective projection of more distant objects to be smaller than that of closer objects. All values of z are allowable except $z = 0$. Points can be behind the center of projection on the negative z axis or between the center of projection and the projection plane.

The transformation of Eq. (6.2) can be expressed as a 4 × 4 matrix:

$$M_{\text{per}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix}. \tag{6.3}$$

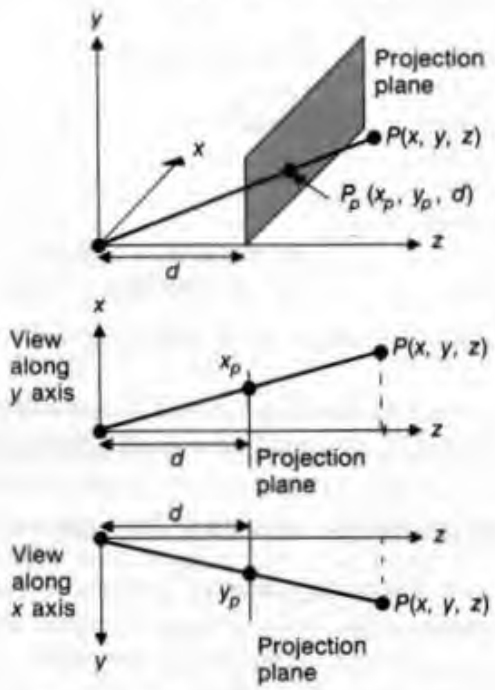


Fig. 6.42 Perspective projection.

Samsung/Google, EX1027, p. 271

Multiplying the point $P = [x \quad y \quad z \quad 1]^T$ by the matrix M_{per} yields the general homogeneous point $[X \quad Y \quad Z \quad W]^T$:

$$\begin{bmatrix} X \\ Y \\ Z \\ W \end{bmatrix} = M_{per} \cdot P = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{6.4}$$

or

$$[X \quad Y \quad Z \quad W]^T = \left[x \quad y \quad z \quad \frac{z}{d}\right]^T. \tag{6.5}$$

Now, dividing by W (which is z/d) and dropping the fourth coordinate to come back to 3D, we have

$$\left(\frac{X}{W}, \frac{Y}{W}, \frac{Z}{W}\right) = (x_p, y_p, z_p) = \left(\frac{x}{z/d}, \frac{y}{z/d}, d\right); \tag{6.6}$$

these equations are the correct results of Eq. (6.1), plus the transformed z coordinate of d, which is the position of the projection plane along the z axis.

An alternative formulation for the perspective projection places the projection plane at $z = 0$ and the center of projection at $z = -d$, as in Fig. 6.43. Similarity of the triangles now gives

$$\frac{x_p}{d} = \frac{x}{z + d}, \qquad \frac{y_p}{d} = \frac{y}{z + d}. \tag{6.7}$$

Multiplying by d, we get

$$x_p = \frac{d \cdot x}{z + d} = \frac{x}{(z/d) + 1}, \qquad y_p = \frac{d \cdot y}{z + d} = \frac{y}{(z/d) + 1}. \tag{6.8}$$

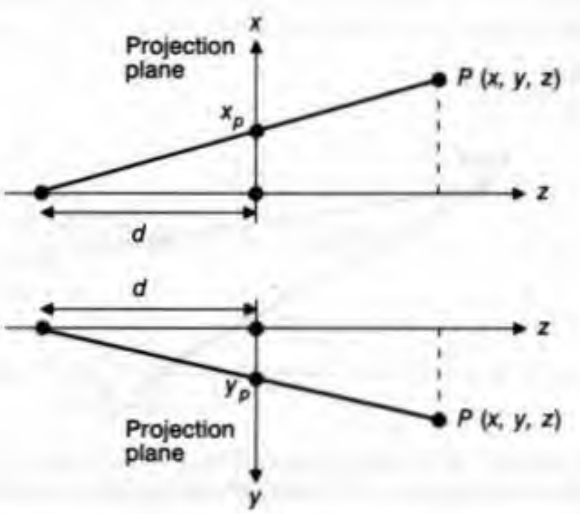


Fig. 6.43 Alternative perspective projection.

Samsung/Google, EX1027, p. 272

The matrix is

$$M'_{per} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1/d & 1 \end{bmatrix}. \tag{6.9}$$

This formulation allows d, the distance to the center of projection, to tend to infinity.

The orthographic projection onto a projection plane at $z = 0$ is straightforward. The direction of projection is the same as the projection-plane normal—the z axis, in this case. Thus, point P projects as

$$x_p = x, \quad y_p = y, \quad z_p = 0. \tag{6.10}$$

This projection is expressed by the matrix

$$M_{ort} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{6.11}$$

Notice that as d in Eq. (6.9) tends to infinity, Eq. (6.9) becomes Eq. (6.11).

M_{per} applies only in the special case in which the center of projection is at the origin; M_{ort} applies only when the direction of projection is parallel to the z axis. A more robust formulation, based on a concept developed in [WEIN87], not only removes these restrictions but also integrates parallel and perspective projections into a single formulation. In Fig. 6.44, the projection of the general point $P = (x, y, z)$ onto the projection plane is $P_p = (x_p, y_p, z_p)$. The projection plane is perpendicular to the z axis at a distance z_p from the origin, and the center of projection (COP) is a distance Q from the point $(0, 0, z_p)$. The direction from $(0, 0, z_p)$ to COP is given by the normalized direction vector (d_x, d_y, d_z). P_p is

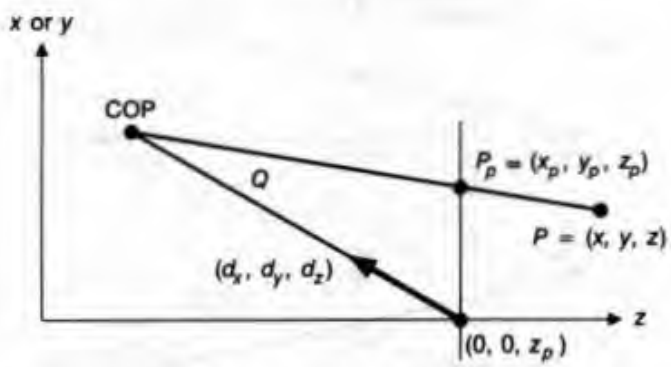


Fig. 6.44 The intersection of the line from COP to $P = (x, y, z)$ with the projection plane at $z = z_p$ is the projection of the point P. The COP is distance Q from the point $(0, 0, z_p)$, in direction (d_x, d_y, d_z).

Samsung/Google, EX1027, p. 273

on the line between COP and P, which can be specified parametrically as

$$\text{COP} + t(P - \text{COP}), \qquad 0 \le t \le 1. \tag{6.12}$$

Rewriting Eq. (6.12) as separate equations for the arbitrary point $P' = (x', y', z')$ on the line, with $\text{COP} = (0, 0, z_p) + Q(d_x, d_y, d_z)$, yields

$$x' = Q\,d_x + (x - Q\,d_x)t, \tag{6.13}$$
$$y' = Q\,d_y + (y - Q\,d_y)t, \tag{6.14}$$
$$z' = (z_p + Q\,d_z) + (z - (z_p + Q\,d_z))t. \tag{6.15}$$

We find the projection P_p of the point P, at the intersection of the line between COP and P with the projection plane, by substituting $z' = z_p$ into Eq. (6.15) and solving for t:

$$t = \frac{z_p - (z_p + Q\,d_z)}{z - (z_p + Q\,d_z)}. \tag{6.16}$$

Substituting this value of t into Eq. (6.13) and Eq. (6.14) to find $x' = x_p$ and $y' = y_p$ yields

$$x_p = \frac{x - z\dfrac{d_x}{d_z} + z_p\dfrac{d_x}{d_z}}{\dfrac{z_p - z}{Q\,d_z} + 1}, \tag{6.17}$$

$$y_p = \frac{y - z\dfrac{d_y}{d_z} + z_p\dfrac{d_y}{d_z}}{\dfrac{z_p - z}{Q\,d_z} + 1}. \tag{6.18}$$

Multiplying the identity $z_p = z_p$ on the right-hand side by a fraction whose numerator and denominator are both

$$\frac{z_p - z}{Q\,d_z} + 1 \tag{6.19}$$

maintains the identity and gives z_p the same denominator as x_p and y_p:

$$z_p = z_p\frac{\dfrac{z_p - z}{Q\,d_z} + 1}{\dfrac{z_p - z}{Q\,d_z} + 1} = \frac{-z\dfrac{z_p}{Q\,d_z} + \dfrac{z_p^2 + z_pQ\,d_z}{Q\,d_z}}{\dfrac{z_p - z}{Q\,d_z} + 1}. \tag{6.20}$$

Now Eqs. (6.17), (6.18), and (6.20) can be rewritten as a 4×4 matrix M_{general} arranged so that the last row of M_{general} multiplied by $[x \;\; y \;\; z \;\; 1]^T$ produces their common

Samsung/Google, EX1027, p. 274

denominator, which is the homogeneous coordinate W and is hence the divisor of X, Y, and Z:

$$M_{general} = \begin{bmatrix} 1 & 0 & -\frac{d_x}{d_z} & z_p\frac{d_x}{d_z} \\ 0 & 1 & -\frac{d_y}{d_z} & z_p\frac{d_y}{d_z} \\ 0 & 0 & -\frac{z_p}{Q\,d_z} & \frac{z_p^2}{Q\,d_z} + z_p \\ 0 & 0 & -\frac{1}{Q\,d_z} & \frac{z_p}{Q\,d_z} + 1 \end{bmatrix}. \tag{6.21}$$

$M_{general}$ specializes to all three of the previously derived projection matrixes M_{per}, M'_{per}, and M_{ort}, given the following values:

	z_p	Q	$[d_x$	d_y	$d_z]$
M_{ort}	0	∞	[0	0	−1]
M_{per}	d	d	[0	0	−1]
M'_{per}	0	d	[0	0	−1]

When $Q \neq \infty$, $M_{general}$ defines a one-point perspective projection. The vanishing point of a perspective projection is calculated by multiplying the point at infinity on the z axis, represented in homogeneous coordinates as $[0 \quad 0 \quad 1 \quad 0]^T$, by $M_{general}$. Taking this product and dividing by W gives

$$x = Q\,d_x, \quad y = Q\,d_y, \quad z = z_p.$$

Given a desired vanishing point (x, y) and a known distance Q to the center of projection, these equations uniquely define $[d_x \quad d_y \quad d_z]$, because $\sqrt{d_x^2 + d_y^2 + d_z^2} = 1$.

Similarly, it is easy to show that, for cavalier and cabinet projections onto the (x, y) plane, with α the angle shown in Figs. 6.10 and 6.11,

	z_p	Q	$[d_x$	d_y	$d_z]$
Cavalier	0	∞	[cosα	sinα	−1]
Cabinet	0	∞	$\left[\frac{\cos\alpha}{2}\right.$	$\frac{\sin\alpha}{2}$	$\left.-1\right]$

In this section, we have seen how to formulate M_{per}, M'_{per}, and M_{ort}, all of which are special cases of the more general $M_{general}$. In all these cases, however, the projection plane is perpendicular to the z axis. In the next section, we remove this restriction and consider the clipping implied by finite view volumes.

6.5 IMPLEMENTING PLANAR GEOMETRIC PROJECTIONS

Given a view volume and a projection, let us consider how the *viewing operation* of clipping and projecting is actually applied. As suggested by the conceptual model for viewing (Fig. 6.1), we could clip lines against the view volume by calculating their intersections with each

Samsung/Google, EX1027, p. 275

of the six planes that define the view volume. Lines remaining after clipping would be projected onto the view plane, by solution of simultaneous equations for the intersection of the projectors with the view plane. The coordinates would then be transformed from 3D world coordinates to 2D device coordinates. However, the large number of calculations required for this process, repeated for many lines, involves considerable computing. Happily, there is a more efficient procedure, based on the divide-and-conquer strategy of breaking down a difficult problem into a series of simpler ones.

Certain view volumes are easier to clip against than is the general one (clipping algorithms are discussed in Section 6.5.3). For instance, it is simple to calculate the intersections of a line with each of the planes of a parallel-projection view volume defined by the six planes

$$x = -1, \quad x = 1, \quad y = -1, \quad y = 1, \quad z = 0, \quad z = -1. \tag{6.22}$$

This is also true of the perspective-projection view volume defined by the planes

$$x = z, \quad x = -z, \quad y = z, \quad y = -z, \quad z = -z_{min}, \quad z = -1. \tag{6.23}$$

These *canonical view volumes* are shown in Fig. 6.45.

Our strategy is to find the *normalizing transformations* N_{par} and N_{per} that transform an arbitrary parallel- or perspective-projection view volume into the parallel and perspective canonical view volumes, respectively. Then clipping is performed, followed by projection into 2D, via the matrices in Section 6.4. This strategy risks investing effort in transforming points that are subsequently discarded by the clip operation, but at least the clipping is easy to do.

Figure 6.46 shows the sequence of processes involved here. We can reduce it to a transform–clip–transform sequence by composing steps 3 and 4 into a single transformation matrix. With perspective projections, a division is also needed to map from homogeneous coordinates back to 3D coordinates. This division is done following the second transformation of the combined sequence. An alternative strategy, clipping in homogeneous coordinates, is discussed in Section 6.5.4.

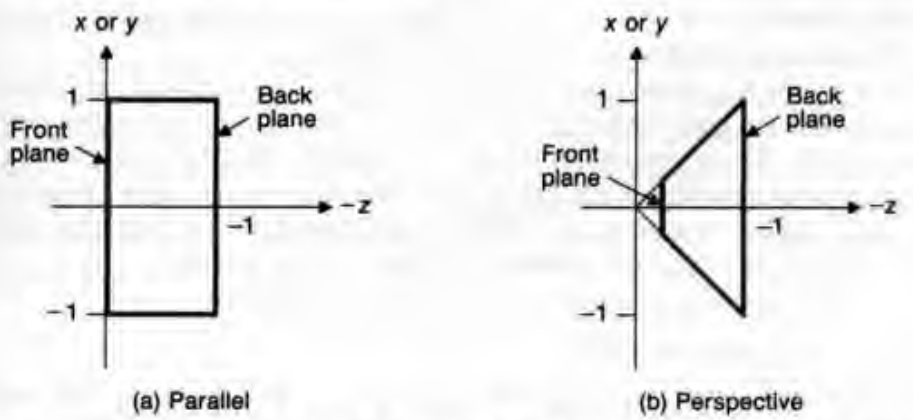


Fig. 6.45 The two canonical view volumes, for the (a) parallel and (b) perspective projections. Note that $-z$ is to the right.

Samsung/Google, EX1027, p. 276

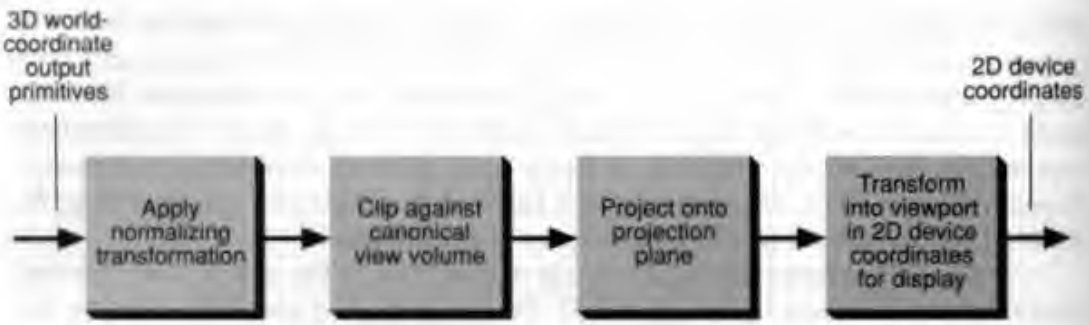


Fig. 6.46 Implementation of 3D viewing.

Readers familiar with PHIGS will notice that the canonical view volumes of Eqs. (6.22) and (6.23) are different than the *default view volumes* of PHIGS: the unit cube from 0 to 1 in x, y, and z for parallel projection, and the pyramid with apex at (0.5, 0.5, 1.0) and sides passing through the unit square from 0 to 1 in x and y on the $z = 0$ plane for perspective projection. The canonical view volumes are defined to simplify the clipping equations and to provide the consistency between parallel and perspective projections discussed in Section 6.5.4. On the other hand, the PHIGS default view volumes are defined to make 2D viewing be a special case of 3D viewing. Exercise 6.26 concerns the PHIGS defaults.

In the next two sections, we derive the normalizing transformations for perspective and parallel projections, which are used as the first step in the transform–clip–transform sequence.

6.5.1 Parallel Projection

In this section, we derive the normalizing transformation N_{par} for parallel projections in order to transform world-coordinate positions such that the view volume is transformed into the canonical view volume defined by Eq. (6.22). The transformed coordinates are clipped against this canonical view volume, and the clipped results are projected onto the $z = 0$ plane, then are transformed into the viewport for display.

Transformation N_{par} is derived for the most general case, the oblique (rather than orthographic) parallel projection. N_{par} thus includes a shear transformation that causes the direction of projection in viewing coordinates to be parallel to z, even though in (u, v, n) coordinates it is not parallel to VPN. By including this shear, we can do the projection onto the $z = 0$ plane simply by setting $z = 0$. If the parallel projection is orthographic, the shear component of the normalization transformation becomes the identity.

The series of transformations that make up N_{par} is as follows:

1. Translate the VRP to the origin
2. Rotate VRC such that the n axis (VPN) becomes the z axis, the u axis becomes the x axis, and the v axis becomes the y axis
3. Shear such that the direction of projection becomes parallel to the z axis
4. Translate and scale into the parallel-projection canonical view volume of Eq. (6.22).

Samsung/Google, EX1027, p. 277

In PHIGS, steps 1 and 2 define the *view-orientation matrix*, and steps 3 and 4 define the *view-mapping matrix*.

Figure 6.47 shows this sequence of transformations as applied to a parallel-projection view volume and to an outline of a house; Fig. 6.48 shows the parallel projection that results.

Step 1 is just the translation $T(-\text{VRP})$. For step 2, we use the properties of orthonormal matrices discussed in Section 5.5 and illustrated in the derivation of Eqs. (5.66) and (5.67). The row vectors of the rotation matrix to perform step 2 are the unit vectors that are rotated by R into the x, y, and z axes. VPN is rotated into the z axis, so

$$R_z = \frac{\text{VPN}}{\|\text{VPN}\|}. \tag{6.24}$$

The u axis, which is perpendicular to VUP and to VPN and is hence the cross-product of the unit vector along VUP and R_z (which is in the same direction as VPN), is rotated into the x axis, so

$$R_x = \frac{\text{VUP} \times R_z}{\|\text{VUP} \times R_z\|}. \tag{6.25}$$

Similarly, the v axis, which is perpendicular to R_z and R_x, is rotated into the y axis, so

$$R_y = R_z \times R_x. \tag{6.26}$$

Hence, the rotation in step 2 is given by the matrix

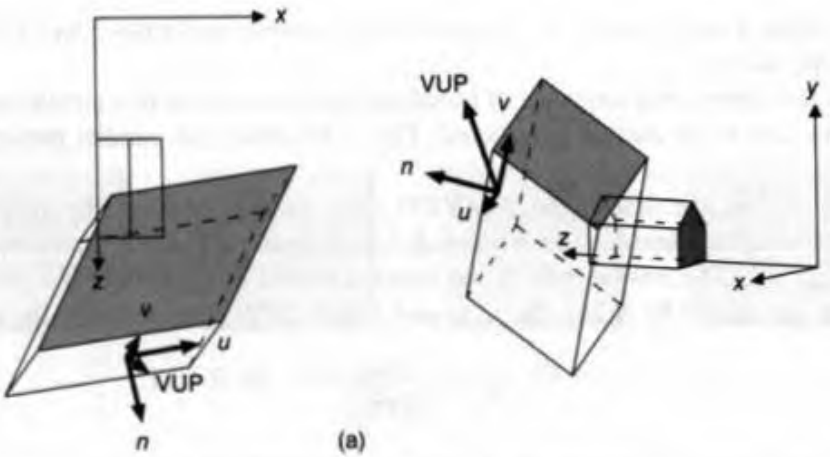
$$R = \begin{bmatrix} r_{1x} & r_{2x} & r_{3x} & 0 \\ r_{1y} & r_{2y} & r_{3y} & 0 \\ r_{1z} & r_{2z} & r_{3z} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{6.27}$$

where r_{1x} is the first element of R_x, and so on.

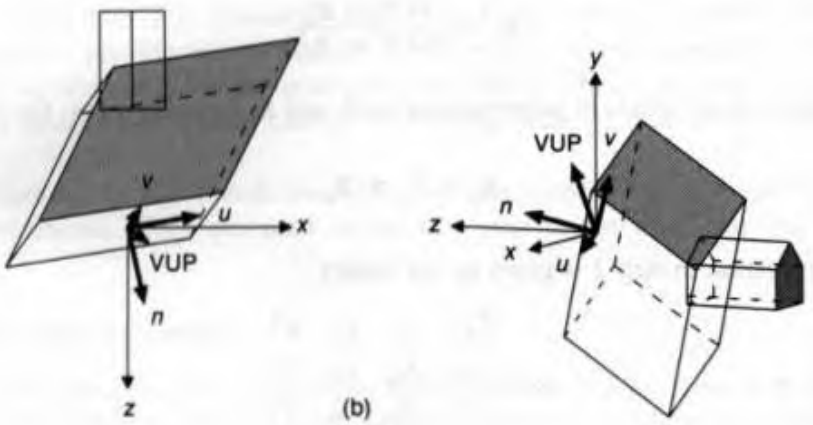
The third step is to shear the view volume along the z axis such that all of its planes are normal to one of the coordinate system axes. We do this step by determining the shear to be applied to the direction of projection (DOP) to make DOP coincident with the z axis. Recall that DOP is the vector from PRP to the center of the window (CW), and that PRP is specified in the VRC system. The first two transformation steps have brought VRC into correspondence with the world-coordinate system, so the PRP is now itself in world coordinates. Hence, DOP is CW − PRP. Given

$$\text{DOP} = \begin{bmatrix} dop_x \\ dop_y \\ dop_z \\ 0 \end{bmatrix}, \quad \text{CW} = \begin{bmatrix} \frac{u_{\max} + u_{\min}}{2} \\ \frac{v_{\max} + v_{\min}}{2} \\ 0 \\ 1 \end{bmatrix}, \quad \text{PRP} = \begin{bmatrix} prp_u \\ prp_v \\ prp_n \\ 1 \end{bmatrix}, \tag{6.28}$$

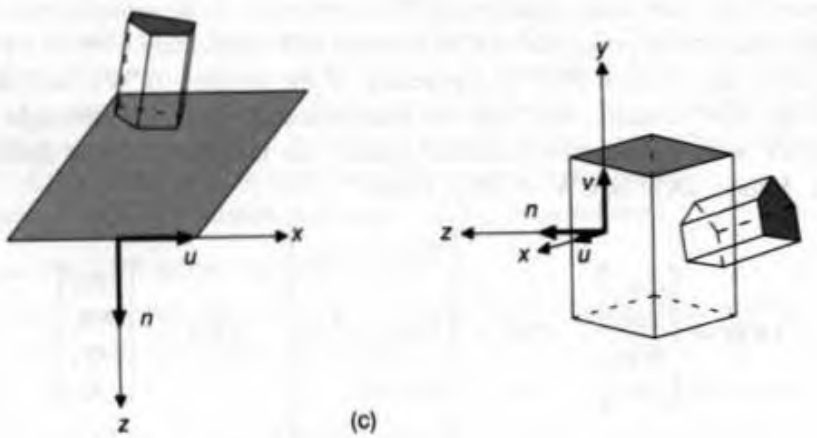
Samsung/Google, EX1027, p. 278



(a)



(b)



(c)

Samsung/Google, EX1027, p. 279

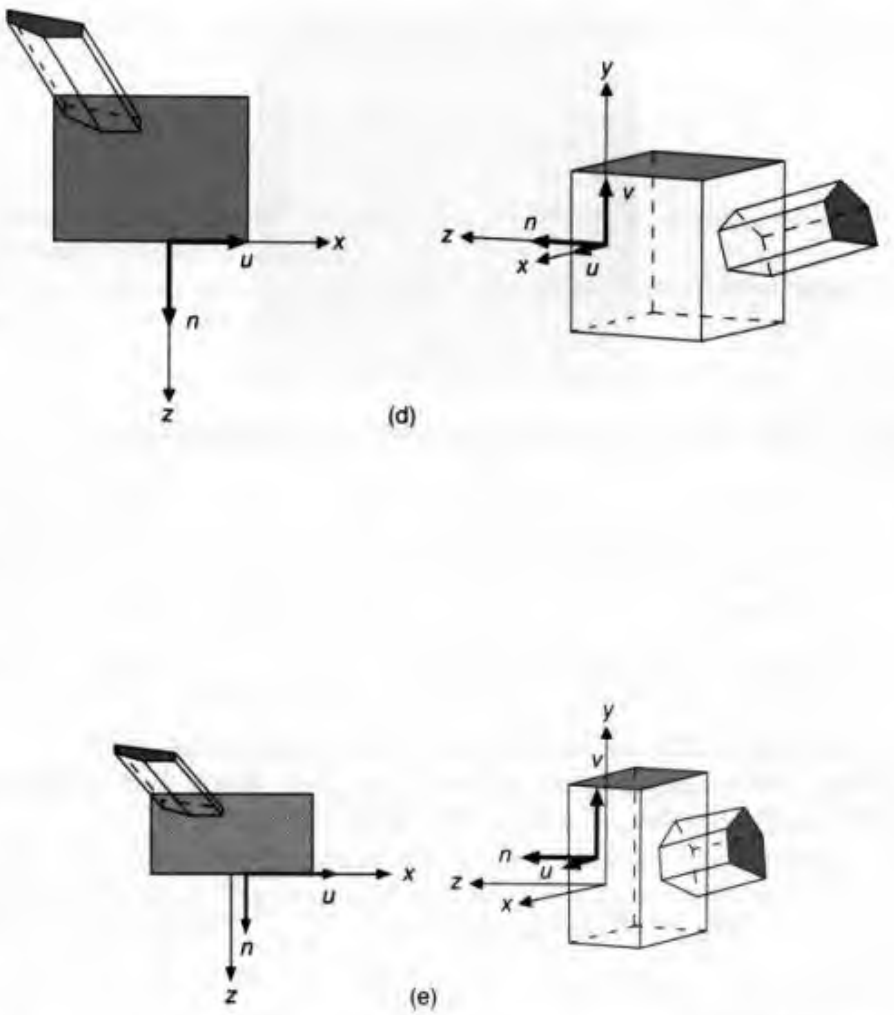


Fig. 6.47 Results at various stages in the parallel-projection viewing pipeline. A top and off-axis parallel projection are shown in each case. (a) The original viewing situation. (b) The VRP has been translated to the origin. (c) The (u, v, n) coordinate system has been rotated to be aligned with the (x, y, z) system. (d) The view volume has been sheared such that the direction of projection (DOP) is parallel to the z axis. (e) The view volume has been translated and scaled into the canonical parallel-projection view volume. The viewing parameters are VRP = (0.325, 0.8, 4.15), VPN = (.227, .267, 1.0), VUP = (.293, 1.0, .227), PRP = (0.6, 0.0, −1.0), Window = (−1.425, 1.0, −1.0, 1.0), $F = 0.0$, $B = -1.75$. (Figures made with program written by Mr. L. Lu, The George Washington University.)

Samsung/Google, EX1027, p. 280



Fig. 6.48 Final parallel projection of the clipped house.

then

$$\begin{aligned}\text{DOP} &= \text{CW} - \text{PRP} \\ &= \left[\frac{u_{max} + u_{min}}{2} \quad \frac{v_{max} + v_{min}}{2} \quad 0 \quad 1\right]^T - [prp_u \quad prp_v \quad prp_n \quad 1].^T \end{aligned} \tag{6.29}$$

Figure 6.49 shows the DOP so specified and the desired transformed DOP′.

The shear can be accomplished with the (x, y) shear matrix from Section 5.6 Eq. (5.47). With coefficients shx_{par} and shy_{par}, the matrix is

$$SH_{par} = SH_{xy}(shx_{par}, shy_{par}) = \begin{bmatrix} 1 & 0 & shx_{par} & 0 \\ 0 & 1 & shy_{par} & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{6.30}$$

As described in Section 5.6, SH_{xy} leaves z unaffected, while adding to x and y the terms $z \cdot shx_{par}$ and $z \cdot shy_{par}$. We want to find shx_{par} and shy_{par} such that

$$\text{DOP}' = [0 \quad 0 \quad dop_z \quad 0]^T = SH_{par} \cdot \text{DOP}. \tag{6.31}$$

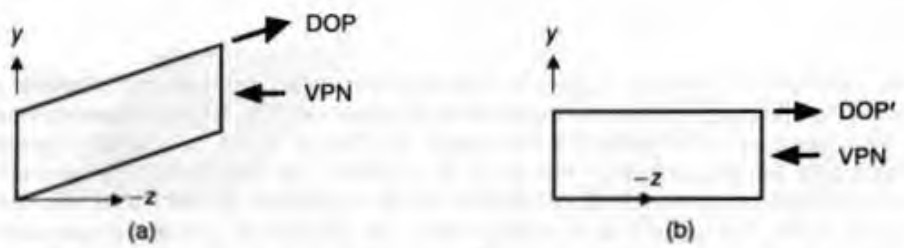


Fig. 6.49 Illustration of shearing using side view of view volume as example. The parallelogram in (a) is sheared into the rectangle in (b); VPN is unchanged because it is parallel to the z axis.

Samsung/Google, EX1027, p. 281

Performing the multiplication of Eq. (6.31) followed by algebraic manipulation shows that the equality occurs if

$$shx_{par} = -\frac{dop_x}{dop_z}, \quad shy_{par} = -\frac{dop_y}{dop_z}. \tag{6.32}$$

Notice that, for an orthographic projection, $dop_x = dop_y = 0$, *so* $shx_{par} = shy_{par} = 0$, and the shear matrix reduces to the identity.

Figure 6.50 shows the view volume after these three transformation steps have been applied. The bounds of the volume are

$$u_{min} \le x \le u_{max}, \qquad v_{min} \le y \le v_{max}, \qquad B \le z \le F; \tag{6.33}$$

here F and B are the distances from VRP along the VPN to the front and back clipping planes, respectively.

The fourth and last step in the process is transforming the sheared view volume into the canonical view volume. We accomplish this step by translating the front center of the view volume of Eq. (6.33) to the origin, then scaling to the $2 \times 2 \times 1$ size of the final canonical view volume of Eq. (6.22). The transformations are

$$T_{par} = T\left(-\frac{u_{max} + u_{min}}{2}, -\frac{v_{max} + v_{min}}{2}, -F\right), \tag{6.34}$$

$$S_{par} = S\left(\frac{2}{u_{max} - u_{min}}, \frac{2}{v_{max} - v_{min}}, \frac{1}{F - B}\right). \tag{6.35}$$

If F and B have not been specified (because front- and back-plane clipping are off), then any values that satisfy $B \le F$ may be used. Values of 0 and 1 are satisfactory.

In summary, we have:

$$N_{par} = S_{par} \cdot T_{par} \cdot SH_{par} \cdot R \cdot T(-\text{VRP}). \tag{6.36}$$

N_{par} transforms an arbitrary parallel-projection view volume into the parallel-projection canonical view volume, and hence permits output primitives to be clipped against the parallel-projection canonical view volume.

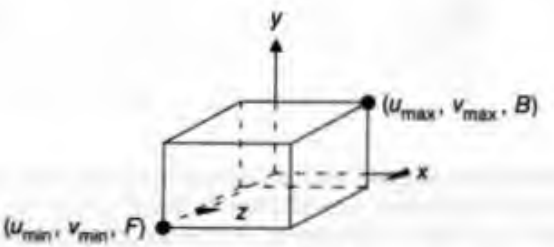


Fig. 6.50 View volume after transformation steps 1 to 3.

Samsung/Google, EX1027, p. 282

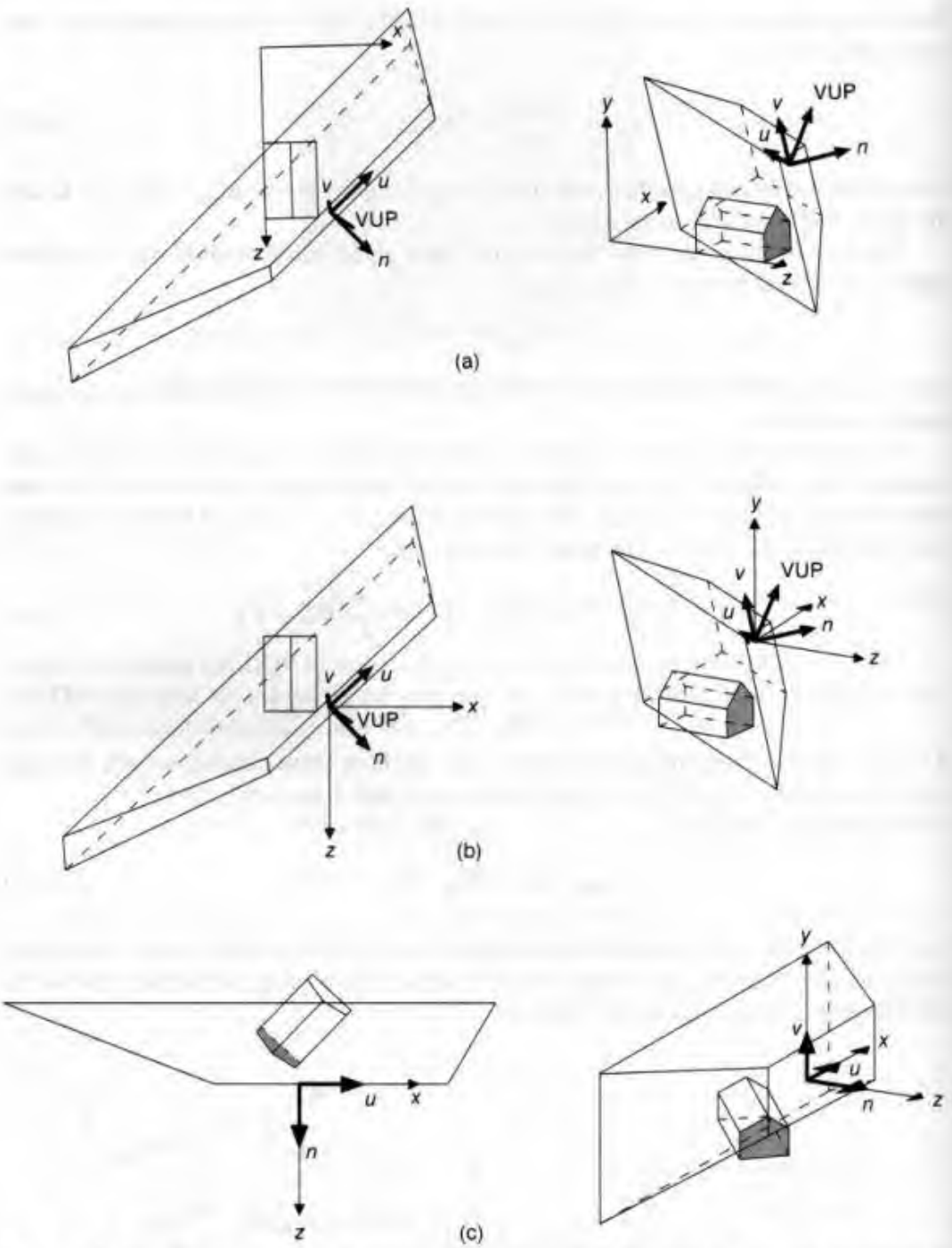
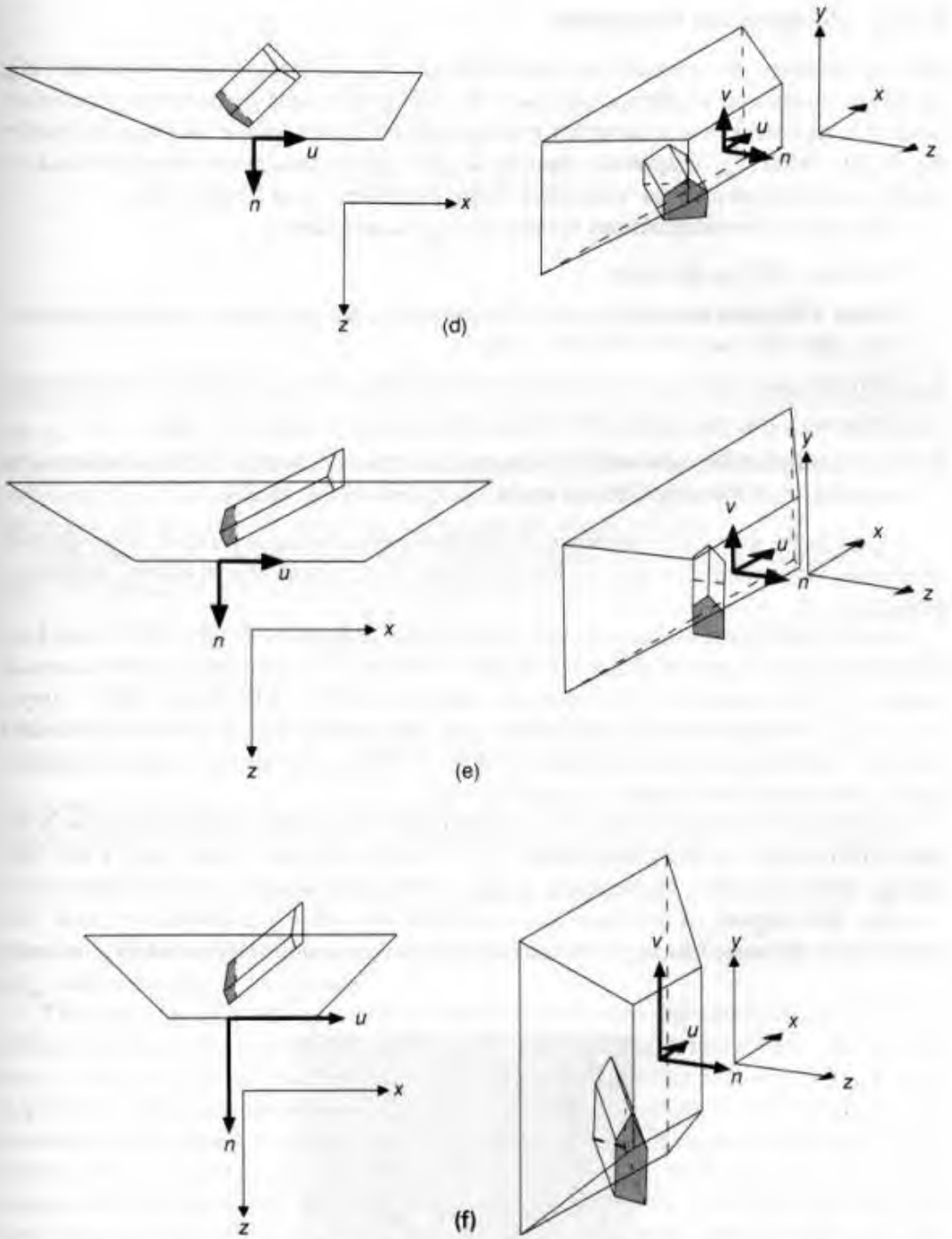


Fig. 6.51 Results at various stages in the perspective-projection viewing pipeline. A top and off-axis parallel projection are shown in each case. (a) The original viewing situation. (b) The VRP has been translated to the origin. (c) The (*u*, *v*, *n*) coordinate system has been rotated to be aligned with the (*x*, *y*, *z*) system. (d) The center of projection (COP) has been translated to the origin. (e) The view volume has been sheared, so the direction of projection (DOP) is parallel to the *z* axis. (f) The view volume

Samsung/Google, EX1027, p. 283



has been scaled into the canonical perspective-projection view volume. The viewing parameters are VRP = (1.0, 1.275, 2.6), VPN = (1.0, 0.253, 1.0), VUP = (0.414, 1.0, 0.253), PRP = (1.6, 0.0, 1.075), Window = (−1.325, 2.25, −0.575, 0.575), $F = 0$, $B = -1.2$. (Figures made with program written by Mr. L. Lu, The George Washington University.)

Samsung/Google, EX1027, p. 284

6.5.2 Perspective Projection

We now develop the normalizing transformation N_{per} for perspective projections. N_{per} transforms world-coordinate positions such that the view volume becomes the perspective-projection canonical view volume, the truncated pyramid with apex at the origin defined by Eq. (6.23). After N_{per} is applied, clipping is done against this canonical volume and the results are projected onto the view plane using M_{per} (derived in Section 6.4).

The series of transformations making up N_{per} is as follows:

1. Translate VRP to the origin
2. Rotate VRC such that the n axis (VPN) becomes the z axis, the u axis becomes the x axis, and the v axis becomes the y axis
3. Translate such that the center of projection (COP), given by the PRP, is at the origin
4. Shear such that the center line of the view volume becomes the z axis
5. Scale such that the view volume becomes the canonical perspective view volume, the truncated right pyramid defined by the six planes of Eq. (6.23).

Figure 6.51 shows this sequence of transformations being applied to a perspective-projection view volume and to a house. Figure 6.52 shows the resulting perspective projection.

Steps 1 and 2 are the same as those for the parallel projection: $R \cdot T(-\text{VRP})$. Step 3 is a translation of the center of projection (COP) to the origin, as required for the canonical perspective view volume. COP is specified relative to VRP in VRC by the PRP = (prp_u, prp_v, prp_n). Viewing-reference coordinates have been transformed into world coordinates by steps 1 and 2, so the specification for COP in VRC is now also in world coordinates. Hence, the translation for step 3 is just $T(-\text{PRP})$.

To compute the shear for step 4, we examine Fig. 6.53, which shows a side view of the view volume after transformation steps 1 to 3. Notice that the center line of the view volume, which goes through the origin and the center of the window, is not the same as the $-z$ axis. The purpose of the shear is to transform the center line into the $-z$ axis. The center line of the view volume goes from PRP (which is now at the origin) to CW, the center



Fig. 6.52 Final perspective projection of the clipped house.

Samsung/Google, EX1027, p. 285

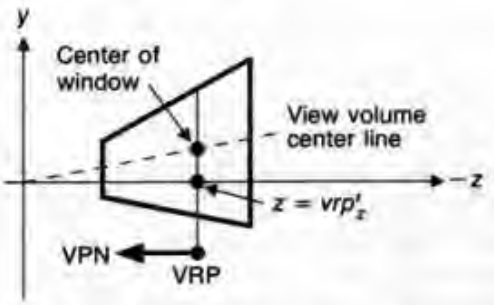


Fig. 6.53 Cross-section of view volume after transformation steps 1 to 3.

of the window. It is hence the same as the direction of projection for the parallel projection, that is, CW − PRP. Therefore the shear matrix is SH_{par}, the same as that for the parallel projection! Another way to think of this is that the translation by −PRP in step 3, which took the center of projection to the origin, also translated CW by −PRP, so, after step 3, the center line of the view volume goes through the origin and CW − PRP.

After the shear is applied, the window (and hence the view volume) is centered on the z axis. The bounds of the window on the projection plane are

$$-\frac{u_{max} - u_{min}}{2} \leq x \leq \frac{u_{max} - u_{min}}{2}, \tag{6.37}$$

$$-\frac{v_{max} - v_{min}}{2} \leq y \leq \frac{v_{max} - v_{min}}{2}.$$

The VRP, which before step 3 was at the origin, has now been translated by step 3 and sheared by step 4. Defining VRP′ as VRP after the transformations of steps 3 and 4,

$$\text{VRP}' = SH_{par} \cdot T(-\text{PRP}) \cdot [0\ 0\ 0\ 1]^T. \tag{6.38}$$

The z component of VRP′, designated as vrp_z', is equal to $-prp_n$, because the (x, y) shear SH_{par} does not affect z coordinates.

The final step is a scaling along all three axes to create the canonical view volume defined by Eq. (6.23), and shown in Fig. 6.54. Thus, scaling is best thought of as being done in two substeps. In the first substep, we scale differentially in x and y, to give the sloped planes bounding the view-volume unit slope. We accomplish this substep by scaling the window so its half-height and half-width are both $-vrp_z'$. The appropriate x and y scale factors are $-2 \cdot vrp_z'/(u_{max} - u_{min})$ and $-2 \cdot vrp_z'/(v_{max} - v_{min})$, respectively. In the second substep, we scale uniformly along all three axes (to maintain the unit slopes) such that the back clipping plane at $z = vrp_z' + B$ becomes the $z = -1$ plane. The scale factor for this substep is $-1/(vrp_z' + B)$. The scale factor has a negative sign so that the scale factor will be positive, since $vrp_z' + B$ is itself negative.

Bringing these two substeps together, we get the perspective scale transformation:

$$S_{per} = S\left(\frac{2\ vrp_z'}{(u_{max} - u_{min})(vrp_z' + B)}, \frac{2\ vrp_z'}{(v_{max} - v_{min})(vrp_z' + B)}, \frac{-1}{vrp_z' + B}\right). \tag{6.39}$$

Samsung/Google, EX1027, p. 286

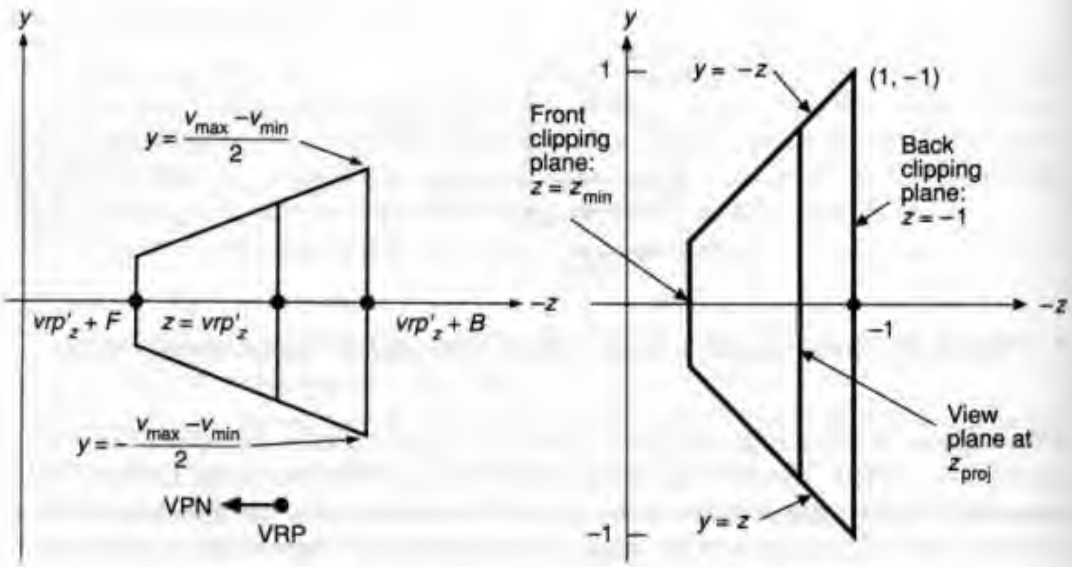


Fig. 6.54 Cross-section of view volume before and after final scaling steps. In this example, F and B have opposite signs, so the front and back planes are on opposite sides of VRP.

Applying the scale to z changes the positions of the projection plane and clipping planes to the new positions:

$$z_{proj} = -\frac{vrp_z'}{vrp_z' + B}, \quad z_{min} = -\frac{vrp_z' + F}{vrp_z' + B}, \quad z_{max} = -\frac{vrp_z' + B}{vrp_z' + B} = -1. \quad (6.40)$$

In summary, the normalizing viewing transformation that takes the perspective-projection view volume into the perspective-projection canonical view volume is

$$N_{per} = S_{per} \cdot SH_{par} \cdot T(-\text{PRP}) \cdot R \cdot T(-\text{VRP}). \quad (6.41)$$

Similarly, recall the normalizing viewing transformation that takes the parallel-projection view volume into the parallel-projection canonical view volume:

$$N_{par} = S_{par} \cdot T_{par} \cdot SH_{par} \cdot R \cdot T(-\text{VRP}). \quad (6.36)$$

These transformations occur in homogeneous space. Under what conditions can we now come back to 3D to clip? So long as we know that $W > 0$. This condition is easy to understand. A negative W implies that, when we divide by W, the signs of Z and z will be opposite. Points with negative Z will have positive z and might be displayed even though they should have been clipped.

When can we be sure that we will have $W > 0$? Rotations, translations, scales, and shears (as defined in Chapter 5) applied to points, lines, and planes will keep $W > 0$; in fact, they will keep $W = 1$. Hence, neither N_{per} nor N_{par} affects the homogeneous coordinate of transformed points, so division by W will not normally be necessary to map back into

Samsung/Google, EX1027, p. 287

3D, and clipping against the appropriate canonical view volume can be performed. After clipping against the perspective-projection canonical view volume, the perspective-projection matrix M_{per}, which involves division, must be applied.

It is possible to get $W < 0$ if output primitives include curves and surfaces that are represented as functions in homogeneous coordinates and are displayed as connected straight-line segments. If, for instance, the sign of the function for W changes from one point on the curve to the next while the sign of X does not change, then X/W will have different signs at the two points on the curve. The rational B-splines discussed in Chapter 11 are an example of such behavior. Negative W can also result from using some transformations other than those discussed in Chapter 5, such as with "fake" shadows [BLIN88].

In the next section, several algorithms for clipping in 3D are discussed. Then, in Section 6.5.4, we discuss how to clip when we cannot ensure that $W > 0$.

6.5.3 Clipping Against a Canonical View Volume in 3D

The canonical view volumes are the unit cube for parallel projections and the truncated right regular pyramid for perspective projections. Both the Cohen–Sutherland and Cyrus–Beck clipping algorithms discussed in Chapter 3 readily extend to 3D.

The extension of the 2D Cohen–Sutherland algorithm for the canonical parallel view volume uses an outcode of 6 bits; a bit is true (1) when the appropriate condition is satisfied:

bit 1—point is above view volume	$y > 1$
bit 2—point is below view volume	$y < -1$
bit 3—point is right of view volume	$x > 1$
bit 4—point is left of view volume	$x < -1$
bit 5—point is behind view volume	$z < -1$
bit 6—point is in front of view volume	$z > 0$

As in 2D, a line is trivially accepted if both endpoints have a code of all zeros, and is trivially rejected if the bit-by-bit logical **and** of the codes is not all zeros. Otherwise, the process of line subdivision begins. Up to six intersections may have to be calculated, one for each side of the view volume.

The intersection calculations use the parametric representation of a line from $P_0(x_0, y_0, z_0)$ to $P_1(x_1, y_1, z_1)$:

$$x = x_0 + t(x_1 - x_0), \tag{6.42}$$

$$y = y_0 + t(y_1 - y_0), \tag{6.43}$$

$$z = z_0 + t(z_1 - z_0)\ 0 \le t \le 1. \tag{6.44}$$

As t varies from 0 to 1, the three equations give the coordinates of all points on the line, from P_0 to P_1.

To calculate the intersection of a line with the $y = 1$ plane of the view volume, we replace the variable y of Eq. (6.43) with 1 and solve for t to find $t = (1 - y_0)/(y_1 - y_0)$. If t is outside the 0 to 1 interval, the intersection is on the infinite line through points P_0 and P_1 but is not on the portion of the line between P_0 and P_1 and hence is not of interest. If t is in

Samsung/Google, EX1027, p. 288

[0, 1], then its value is substituted into the equations for x and z to find the intersection's coordinates:

$$x = x_0 + \frac{(1 - y_0)(x_1 - x_0)}{y_1 - y_0}, \quad z = z_0 + \frac{(1 - y_0)(z_1 - z_0)}{y_1 - y_0}. \tag{6.45}$$

The algorithm uses outcodes to make the t in [0, 1] test unnecessary.

The outcode bits for clipping against the canonical perspective view volume are as follows:

bit 1—point is above view volume	$y > -z$
bit 2—point is below view volume	$y < z$
bit 3—point is right of view volume	$x > -z$
bit 4—point is left of view volume	$x < z$
bit 5—point is behind view volume	$z < -1$
bit 6—point is in front of view volume	$z > z_{min}$

Calculating the intersections of lines with the sloping planes is simple. On the $y = z$ plane, for which Eq. (6.43) must be equal to Eq. (6.44), $y_0 + t(y_1 - y_0) = z_0 + t(z_1 - z_0)$. Then,

$$t = \frac{z_0 - y_0}{(y_1 - y_0) - (z_1 - z_0)}. \tag{6.46}$$

Substituting t into Eqs. (6.42) and (6.43) for x and y gives

$$x = x_0 + \frac{(x_1 - x_0)(z_0 - y_0)}{(y_1 - y_0) - (z_1 - z_0)}, \quad y = y_0 + \frac{(y_1 - y_0)(z_0 - y_0)}{(y_1 - y_0) - (z_1 - z_0)}. \tag{6.47}$$

We know that $z = y$. The reason for choosing this canonical view volume is now clear: The unit slopes of the planes make the intersection computations simpler than would arbitrary slopes.

The Cyrus–Beck clipping algorithm was formulated for clipping a line against a general convex 3D polyhedron, and specialized to the 3D viewing pyramid [CYRU78]. Liang and Barsky later independently developed a more efficient and more specialized version for upright 2D and 3D clip regions [LIAN84]. In 2D, at most four values of t are computed, one for each of the four window edges; in 3D, at most six values. The values are kept or discarded on the basis of exactly the same criteria as those for 2D until exactly two values of t remain (the values will be in the interval [0, 1]). For any value of t that is neither 0 nor 1, Eqs. (6.42), (6.43), and (6.44) are used to compute the values of x, y, and z. In the case of the parallel-projection canonical view volume, we do the extension to 3D by working with the code given in Section 3.12.4. In the case of the perspective-projection canonical view volume, new values of N_i, P_{Ei}, $P_0 - P_{Ei}$, and t need to be found for the six planes; they are given in Table 6.1. The actual code developed by Liang and Barsky but modified to be consistent with our variable names and to use outward rather than inward normals is given in Fig. 6.55. The correspondence between the numerator and denominator of the fractions in the last column of Table 6.1 and the terms in the code can be seen by inspection. The sign of $-N_i \cdot D$, which is the denominator of t, is used by procedure CLIPt in Fig. 6.55 to determine whether the intersection is potentially entering or potentially leaving. Hence, in

Samsung/Google, EX1027, p. 289

TABLE 6.1 KEY VARIABLES AND EQUATIONS FOR CYRUS–BECK 3D CLIPPING AGAINST THE CANONICAL PERSPECTIVE PROJECTION VIEW VOLUME*

Clip edge	Outward normal N_i	Point on edge, P_{Ei}	$P_0 - P_{Ei}$	$t = \frac{N_i \cdot (P_0 - P_{Ei})}{-N_i \cdot D}$		
right: $x = -z$	$(1, 0, 1)$	$(x, y, -x)$	$(x_0 - x, y_0 - y, z_0 + x)$	$\frac{(x_0 - x) + (z_0 + x)}{-(dx + dz)}$	=	$\frac{x_0 + z_0}{-dx - dz}$
left: $x = z$	$(-1, 0, 1)$	(x, y, x)	$(x_0 - x, y_0 - y, z_0 - x)$	$\frac{-(x_0 - x) + (z_0 - x)}{dx - dz}$	=	$\frac{-x_0 + z_0}{dx - dz}$
bottom: $y = z$	$(0, -1, 1)$	(x, y, y)	$(x_0 - x, y_0 - y, z_0 - y)$	$\frac{-(y_0 - y) + (z_0 - y)}{dy - dz}$	=	$\frac{-y_0 + z_0}{dy - dz}$
top: $y = -z$	$(0, 1, 1)$	$(x, y, -y)$	$(x_0 - x, y_0 - y, z_0 + y)$	$\frac{(y_0 - y) + (z_0 + y)}{-dy - dz}$	=	$\frac{y_0 + z_0}{-dy - dz}$
front: $z = z_{min}$	$(0, 0, 1)$	(x, y, z_{min})	$(x_0 - x, y_0 - y, z_0 - z_{min})$	$\frac{(z_0 - z_{min})}{-dz}$	=	$\frac{z_0 - z_{min}}{-dz}$
back: $z = -1$	$(0, 0, -1)$	$(x, y, -1)$	$(x_0 - x, y_0 - y, z_0 + 1)$	$\frac{-(z_0 + 1)}{dz}$	=	$\frac{-z_0 - 1}{dz}$

*The variable D, which is $P_1 - P_0$, is represented as (dx, dy, dz). The exact coordinates of the point P_{Ei} on each edge are irrelevant to the computation, so they have been denoted by variables x, y, and z. For a point on the right edge, $z = -x$, as indicated in the first row, third entry.

Samsung/Google, EX1027, p. 290

```
void Clip3D (double *x0, double *y0, double *z0, double *x1, double *y1, double *z1,
    double *zmin, boolean *accept)
{
    double tmin = 0.0, tmax = 1.0;
    double dx = *x1 - *x0, dz = *z1 - *z0;
    *accept = FALSE;        /* Assume initially that none of the line is visible */
    if (CLIPt (-dx - dz, *x0 + *z0, &tmin, &tmax))                    /* Right side */
        if (CLIPt (dx - dz, -*x0 + *z0, &tmin, &tmax)) {              /* Left side */
            /* If get this far, part of line is in -z <= x <= z */
            double dy = *y1 - *y0;
            if (CLIPt (dy - dz, -*y0 + *z0, &tmin, &tmax))            /* Bottom */
                if (CLIPt (-dy - dz, *y0 + *z0, &tmin, &tmax))        /* Top */
                    /* If get this far, part of line is in -z <= x <= z, -z <= y <= z */
                    if (CLIPt (-dz, *z0 - *zmin, &tmin, &tmax))       /* Front */
                        if (CLIPt (dz, -*z0 - 1, &tmin, &tmax)) {     /* Back */
                            /* If get here, part of line is visible in -z <= x <= z, */
                            /* -z <= y <= z, -1 <= z <= zmin */
                            *accept = TRUE;                           /* Part of line is visible */
                            /* If endpoint 1 (t = 1) is not in the region, compute intersection */
                            if (tmax < 1.0) {                         /* Eqs. 6.37 to 6.39 */
                                *x1 = *x0 + tmax * dx;
                                *y1 = *y0 + tmax * dy;
                                *z1 = *z0 + tmax * dz;
                            }
                            /* If endpoint 0 (t = 0) is not in the region, compute intersection */
                            if (tmin > 0.0) {                         /* Eqs. 6.37 to 6.39 */
                                *x0 += tmin * dx;
                                *y0 += tmin * dy;
                                *z0 += tmin * dz;
                            }
                        } /* Calculating intersection */
        }
} /* Clip3D */
```

Fig. 6.55 The Liang–Barsky 2D clipping algorithm, extended to the 3D canonical perspective-projection view volume. The code is from [LIAN84]. The function CLIPt is in Chapter 3, Fig. 3.45.

the algebraic simplifications done in the table, the sign of the denominator of t is always maintained.

The Sutherland–Hodgman polygon-clipping algorithm can be readily adapted to 3D. We use six clipping planes rather than four, by making six calls to S_H_CLIP (Chapter 3) instead of four.

Once the clipping is done, the surviving output primitives are projected onto the projection plane with either M_{ort} or M_{per}, and are transformed for display into the physical-device-coordinate viewport.

Samsung/Google, EX1027, p. 291

6.5.4 Clipping in Homogeneous Coordinates

There are two reasons to clip in homogeneous coordinates. The first has to do with efficiency: It is possible to transform the perspective-projection canonical view volume into the parallel-projection canonical view volume, so a single clip procedure, optimized for the parallel-projection canonical view volume, can always be used. However, the clipping must be done in homogeneous coordinates to ensure correct results. A single clip procedure is typically provided in hardware implementations of the viewing operation (Chapter 18). The second reason is that points that can occur as a result of unusual homogeneous transformations and from use of rational parametric splines (Chapter 11) can have negative W and can be clipped properly in homogeneous coordinates but not in 3D.

With regard to clipping, it can be shown that the transformation from the perspective-projection canonical view volume to the parallel-projection canonical view volume is

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{1}{1+z_{min}} & \frac{-z_{min}}{1+z_{min}} \\ 0 & 0 & -1 & 0 \end{bmatrix}, \qquad z_{min} \neq -1. \tag{6.48}$$

Recall from Eq. (6.40) that $z_{min} = -(\text{vrp}'_z + F)/(\text{vrp}'_z + B)$, and from Eq. (6.38) that $\text{VRP}' = SH_{par} \cdot T(-\text{PRP}) \cdot [0\ 0\ 0\ 1]^T$. Figure 6.56 shows the results of applying M to the perspective-projection canonical view volume.

The matrix M is integrated with the perspective-projection normalizing transformation N_{per}:

$$N'_{per} = M \cdot N_{per} = M \cdot S_{per} \cdot SH_{par} \cdot T(-\text{PRP}) \cdot R \cdot T(-\text{VRP}). \tag{6.49}$$

By using N'_{per} instead of N_{per} for perspective projections, and by continuing to use N_{par} for parallel projections, we can clip against the parallel-projection canonical view volume rather than against the perspective-projection canonical view volume.

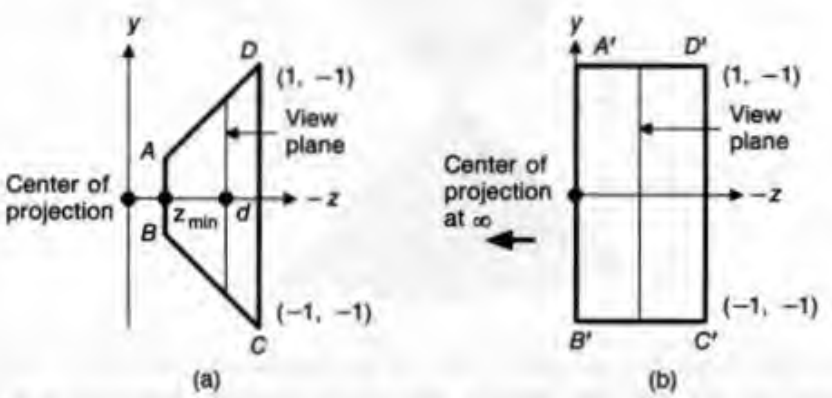


Fig 6.56 Side views of normalized perspective view volume before (a) and after (b) application of matrix M.

Samsung/Google, EX1027, p. 292

The 3D parallel-projection view volume is defined by $-1 \le x \le 1, -1 \le y \le 1, -1 \le z \le 0$. We find the corresponding inequalities in homogeneous coordinates by replacing x by X/W, y by Y/W, and z by Z/W, which results in

$$-1 \le X/W \le 1, \; -1 \le Y/W \le 1, \; -1 \le Z/W \le 0. \tag{6.50}$$

The corresponding plane equations are

$$X = -W, \; X = W, \; Y = -W, \; Y = W, \; Z = -W, \; Z = 0. \tag{6.51}$$

To understand how to use these limits and planes, we must consider separately the cases of $W > 0$ and $W < 0$. In the first case, we can multiply the inequalities of Eq. (6.50) by W without changing the sense of the inequalities. In the second case, the multiplication changes the sense. This result can be expressed as

$$W > 0: \; -W \le X \le W, \; -W \le Y \le W, \; -W \le Z \le 0, \tag{6.52}$$

$$W < 0: \; -W \ge X \ge W, \; -W \ge Y \ge W, \; -W \ge Z \ge 0. \tag{6.53}$$

In the case at hand—that of clipping ordinary lines and points—only the region given by Eq. (6.52) needs to be used, because prior to application of M, all visible points have $W > 0$ (normally $W = 1$).

As we shall see in Chapter 11, however, it is sometimes desirable to represent points directly in homogeneous coordinates with arbitrary W coordinates. Hence, we might have a $W < 0$, meaning that clipping must be done against the regions given by Eqs. (6.52) and (6.53). Figure 6.57 shows these as region A and region B, and also shows why both regions must be used.

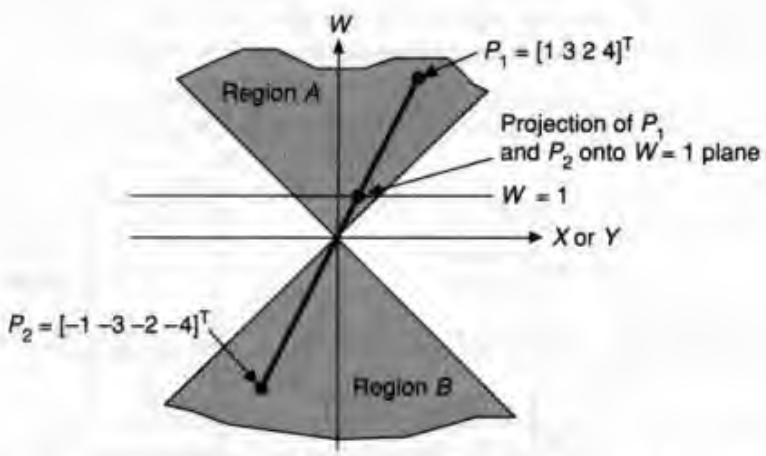


Fig. 6.57 The points P_1 and P_2 both map into the same point on the $W = 1$ plane, as do all other points on the line through the origin and the two points. Clipping in homogeneous coordinates against just region A will incorrectly reject P_2.

Samsung/Google, EX1027, p. 293

The point $P_1 = [1 \quad 3 \quad 2 \quad 4]^T$ in region A transforms into the 3D point $(\frac{1}{4}, \frac{3}{4}, \frac{2}{4})$, which is in the canonical view volume $-1 \leq x \leq 1$, $-1 \leq y \leq 1$, $-1 \leq z \leq 0$. The point $P_2 = -P_1 = [-1 \quad -3 \quad -2 \quad -4]^T$, which is *not* in region A but *is* in region B, transforms into the same 3D point as P_1; namely, $(\frac{1}{4}, \frac{3}{4}, \frac{2}{4})$. If clipping were only to region A, then P_2 would be discarded incorrectly. This possibility arises because the homogeneous coordinate points P_1 and P_2 differ by a constant multiplier (-1), and we know that such homogeneous points correspond to the same 3D point (on the $W = 1$ plane of homogeneous space).

There are two solutions to this problem of points in region B. One is to clip all points twice, once against each region. But doing two clips is expensive. A better solution is to negate points, such as P_2, with negative W, and then to clip them. Similarly, we can clip properly a line whose endpoints are both in region B of Fig. 6.57 by multiplying both endpoints by -1, to place the points in region A.

Another problem arises with lines such as P_1P_2, shown in Fig. 6.58, whose endpoints have opposite values of W. The projection of the line onto the $W = 1$ plane is two segments, one of which goes to $+\infty$, the other to $-\infty$. The solution now is to clip twice, once against each region, with the possibility that each clip will return a visible line segment. A simple way to do this is to clip the line against region A, to negate both endpoints of the line, and to clip again against region A. This approach preserves one of the original purposes of clipping in homogeneous coordinates: using a single clip region. Interested readers are referred to [BLIN78a] for further discussion.

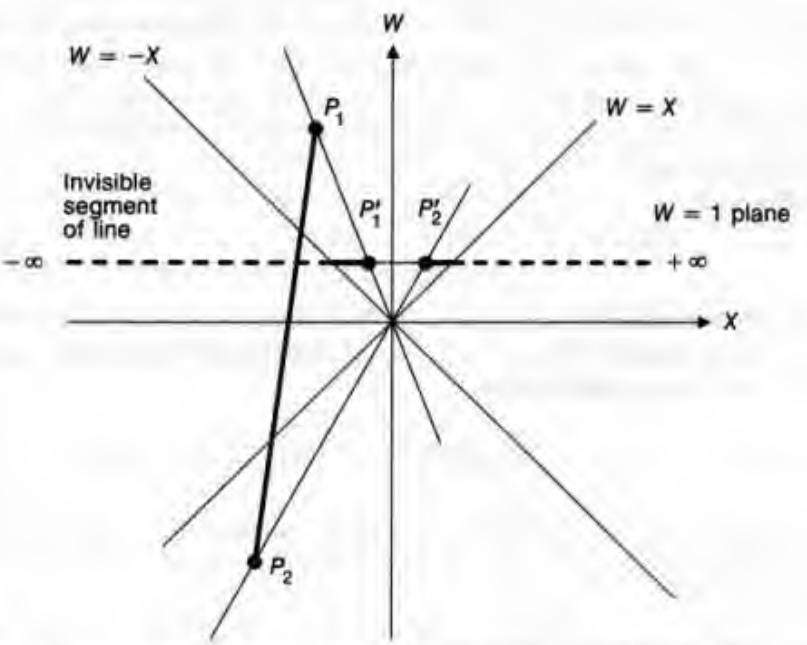


Fig. 6.58 The line P_1P_2 projects onto two line segments, one from P_2' to $+\infty$, the other from P_1' to $-\infty$ (shown as solid thick lines where they are in the clip region, and as dashed thick lines where they are outside the clip region). The line must be clipped twice, once against each region.

Samsung/Google, EX1027, p. 294

Given Eq. (6.51), the Cohen–Sutherland or Cyrus–Beck algorithm can be used for the actual clipping. ([LIAN84] gives code for the Cyrus–Beck approach; see also Exercise 6.25.) The only difference is that the clipping is in 4D, as opposed to 3D.

6.5.5 Mapping into a Viewport

Output primitives are clipped in the normalized projection coordinate system, which is also called the 3D screen coordinate system. We will assume for this discussion that the canonical parallel projection view volume has been used for clipping (the perspective projection M transforms the perspective projection view volume into the parallel projection view volume if this assumption is incorrect). Hence the coordinates of all output primitives that remain are in the view volume $-1 \leq x \leq 1$, $-1 \leq y \leq 1$, $-1 \leq z \leq 0$.

The PHIGS programmer specifies a 3D viewport into which the contents of this view volume are mapped. The 3D viewport is contained in the unit cube $0 \leq x \leq 1$, $0 \leq y \leq 1$, $0 \leq z \leq 1$. The $z = 1$ front face of the unit cube is mapped into the largest square that can be inscribed on the display screen. We assume that the lower left corner of the square is at (0, 0). For example, on a display device with a horizontal resolution of 1024 and a vertical resolution of 800, the square is the region $0 \leq x \leq 799$, $0 \leq y \leq 799$. Points in the unit cube are displayed by discarding their z coordinate. Hence the point (0.5, 0.75, 0.46) would be displayed at the device coordinates (400, 599). In the case of visible-surface determination (Chapter 15), the z coordinate of each output primitive is used to determine which primitives are visible and which are obscured by other primitives with larger z.

Given a 3D viewport within the unit cube with coordinates $x_{v,min}$, $x_{v,max}$, and so forth, the mapping from the canonical parallel projection view volume into the 3D viewport can be thought of as a three-step process. In the first step, the canonical parallel projection view volume is translated so its corner, $(-1, -1, -1)$, becomes the origin. This is effected by the translation $T(1, 1, 1)$. Next, the translated view volume is scaled into the size of the 3D viewport, with the scale

$$S\left(\frac{x_{v,max} - x_{v,min}}{2}, \frac{y_{v,max} - y_{v,min}}{2}, \frac{z_{v,max} - z_{v,min}}{1}\right).$$

Finally, the properly scaled view volume is translated to the lower-left corner of the viewport by the translation $T(x_{v,min}, y_{v,min}, z_{v,min})$. Hence the composite canonical view volume to 3D viewport transformation is

$$M_{VV3DV} = T(x_{v,min}, y_{v,min}, z_{v,min}) \cdot S\left(\frac{x_{v,max} - x_{v,min}}{2}, \frac{y_{v,max} - y_{v,min}}{2}, \frac{z_{v,max} - z_{v,min}}{1}\right) \cdot T(1, 1, 1) \tag{6.54}$$

Note that this is similar to, but not the same as, the window to viewport transformation M_{WV} developed in Section 5.4.

6.5.6 Implementation Summary

There are two generally used implementations of the overall viewing transformation. The first, depicted in Fig. 6.46 and discussed in Sections 6.5.1 through 6.5.3, is appropriate when

Samsung/Google, EX1027, p. 295

output primitives are defined in 3D and the transformations applied to the output primitives never create a negative W. Its steps are as follows:

1. Extend 3D coordinates to homogeneous coordinates
2. Apply normalizing transformation N_{par} or N_{per}
3. Divide by W to map back to 3D (in some cases, it is known that $W = 1$, so the division is not needed)
4. Clip in 3D against the parallel-projection or perspective-projection canonical view volume, whichever is appropriate
5. Extend 3D coordinates to homogeneous coordinates
6. Perform parallel projection using M_{ort}, Eq. (6.11), or perform perspective projection, using M_{per}, Eq. (6.3) with $d = -1$ (because the canonical view volume lies along the $-z$ axis)
7. Translate and scale into device coordinates using Eq. (6.54)
8. Divide by W to map from homogeneous to 2D coordinates; the division effects the perspective projection.

Steps 6 and 7 are performed by a single matrix multiplication, and correspond to stages 3 and 4 in Fig. 6.46.

The second way to implement the viewing operation is required whenever output primitives are defined in homogeneous coordinates and might have $W < 0$, when the transformations applied to the output primitives might create a negative W, or when a single clip algorithm is implemented. As discussed in Section 6.5.4, its steps are as follows:

1. Extend 3D coordinates to homogeneous coordinates
2. Apply normalizing transformation N_{par} or N_{per}' (which includes M, Eq. (6.48))
3. If $W > 0$, clip in homogeneous coordinates against the volume defined by Eq. 6.52; else, clip in homogeneous coordinates against the two view volumes defined by Eqs. (6.52) and (6.53)
4. Translate and scale into device coordinates using Eq. (6.54)
5. Divide by W to map from homogeneous coordinates to 2D coordinates; the division effects the perspective projection.

6.6 COORDINATE SYSTEMS

Several different coordinate systems have been used in Chapters 5 and 6. In this section, we summarize all the systems, and also discuss their relationships to one another. Synonyms used in various references and graphics subroutine packages are also given. Figure 6.59 shows the progression of coordinate systems, using the terms generally used in this text; in any particular graphics subroutine package, only some of the coordinate systems are actually used. We have chosen names for the various coordinate systems to reflect common usage; some of the names therefore are not logically consistent with one another. Note that the term *space* is sometimes used as a synonym for *system*.

Samsung/Google, EX1027, p. 296

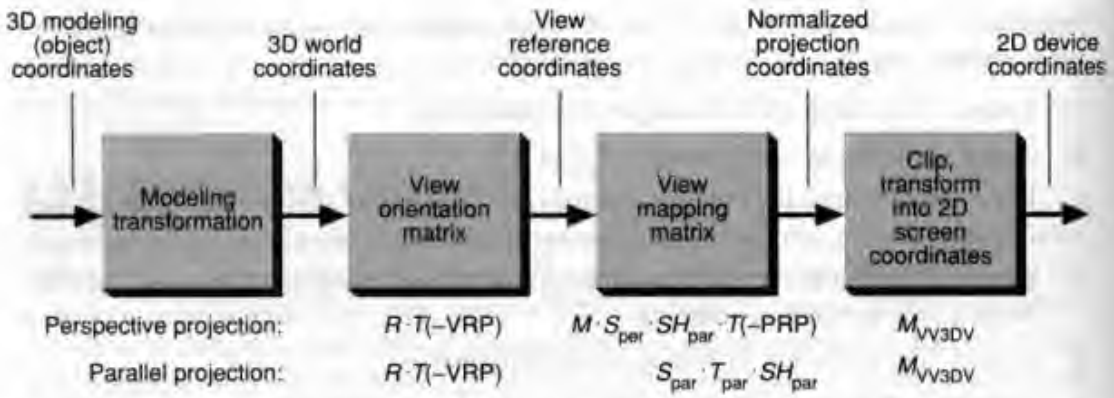


Fig. 6.59 Coordinate systems and how they relate to one another. The matrices underneath each stage effect the transformation applied at that stage for the perspective and parallel projections.

Starting with the coordinate system that is furthest removed from the actual display device, on the left of Fig. 6.59, individual objects are defined in an *object-coordinate system*. PHIGS calls this the *modeling-coordinate system*; the term *local coordinate system* is also commonly used. As we shall discuss further in Chapter 7, there is often a hierarchy of modeling coordinate systems.

Objects are transformed into the *world-coordinate system*, the system in which a scene or complete object is represented in the computer, by the *modeling transformation*. This system is sometimes called the *problem-coordinate system* or *application-coordinate system*.

The *view-reference coordinate system* is used by PHIGS as a coordinate system to define a view volume. It is also called the (u, v, n) system, or the (u, v, VPN) system. The Core system [GSPC79] used a similar, but unnamed, left-handed system. The left-handed system is used so that, with the eye or camera at the origin looking toward $+z$, increasing values of z are farther away from the eye, x is to the right, and y is up.

Other packages, such as Pixar's RenderMan [PIXA88], place constraints on the view-reference coordinate system, requiring that the origin be at the center of projection and that the view plane normal be the z axis. We call this the *eye-coordinate system*; Renderman and some other systems use the term *camera-coordinate system*. Referring back to Section 6.5, the first three steps of the perspective-projection normalizing transformation convert from the world-coordinate system into the eye-coordinate system. The eye-coordinate system is sometimes left-handed.

From eye coordinates, we next go to the *normalized-projection coordinate system*, or *3D screen coordinates*, the coordinate system of the parallel-projection canonical view volume (and of the perspective-projection canonical view volume after the perspective transformation). The Core system calls this system *3D normalized device coordinates*. Sometimes, the system is called *3D logical device coordinates*. The term *normalized* generally means that all the coordinate values are in either the interval $[0, 1]$ or $[-1, 1]$, whereas the term *logical* generally means that coordinate values are in some other

Samsung/Google, EX1027, p. 297

prespecified range, such as [0, 1023], which is typically defined to correspond to some widely available device's coordinate system. In some cases, this system is not normalized.

Projecting from 3D into 2D creates what we call the *2D device coordinate system*, also called the *normalized device-coordinate system*, the *image-coordinate system* by [SUTH74], or the *screen-coordinate system* by RenderMan. Other terms used include *screen coordinates*, *device coordinates*, *2D device coordinates*, *physical device coordinates* (in contrast to the logical device coordinates mentioned previously). RenderMan calls the physical form of the space *raster coordinates*.

Unfortunately, there is no single standard usage for many of these terms. For example, the term *screen-coordinate system* is used by different authors to refer to the last three systems discussed, covering both 2D and 3D coordinates, and both logical and physical coordinates.

EXERCISES

6.1 Write a program that accepts a viewing specification, calculates either N_{par} or N_{per}, and displays the house.

6.2 Program 3D clipping algorithms for parallel and perspective projections.

6.3 Show that, for a parallel projection with $F = -\infty$ and $B = +\infty$, the result of clipping in 3D and then projecting to 2D is the same as the result of projecting to 2D and then clipping in 2D.

6.4 Show that, if all objects are in front of the center of projection and if $F = -\infty$ and $B = +\infty$, then the result of clipping in 3D against the perspective-projection canonical view volume followed by perspective projection is the same as first doing a perspective projection into 2D and then clipping in 2D.

6.5 Verify that S_{per} (Section 6.5.2) transforms the view volume of Fig. 6.54(a) into that of Fig. 6.54(b).

6.6 Write the code for 3D clipping against the unit cube. Generalize the code to clip against any rectangular solid with faces normal to the principal axes. Is the generalized code more or less efficient than that for the unit-cube case?

6.7 Write the code for 3D clipping against the perspective-projection canonical view volume. Now generalize to the view volume defined by

$$-a \cdot z_v \le x_v \le b \cdot z_v, \quad -c \cdot z_v \le y_v \le d \cdot z_v, \quad z_{min} \le z_v \le z_{max}.$$

This is the general form of the view volume after steps 1 to 4 of the perspective normalizing transformation. Which case is more efficient?

6.8 Write the code for 3D clipping against a general six-faced polyhedral view volume whose faces are defined by

$$A_i x + B_i y + C_i z + D_i = 0, \ 1 \le i \le 6.$$

Compare the computational effort needed with that required for each of the following:

a. Clipping against either of the canonical view volumes
b. Applying N_{par} and then clipping against the unit cube.

Samsung/Google, EX1027, p. 298

6.9 Consider a line in 3D going from the world-coordinate points P_1 (6, 10, 3) to P_2 (−3, −5, 2) and a semi-infinite viewing pyramid in the region $-z \le x \le z$, $-z \le y \le z$, which is bounded by the planes $z = +x$, $z = -x$, $z = +y$, $z = -y$. The projection plane is at $z = 1$.

a. Clip the line in 3D (using parametric line equations), then project it onto the projection plane. What are the clipped endpoints on the plane?

b. Project the line onto the plane, then clip the lines using 2D computations. What are the clipped endpoints on the plane?

(Hint: If your answers to (a) and (b) are not identical, try again!)

6.10 Show what happens when an object "behind" the center of projection is projected by M_{per} and then clipped. Your answer should demonstrate why, in general, one cannot project and then clip.

6.11 Consider the 2D viewing operation, with a rotated window. Devise a normalized transformation to transform the window into the unit square. The window is specified by u_{min}, v_{min}, u_{max}, v_{max} in the VRC coordinate system, as in Fig. 6.60. Show that this transformation is the same as that for the general 3D N_{par}, when the projection plane is the (x, y) plane and VUP has an x component of $-\sin\theta$ and a y component of $\cos\theta$ (i.e., the parallel projection of VUP onto the view plane is the v axis).

6.12 The matrix M_{per} in Section 6.4 defines a one-point perspective projection. What is the form of the 4×4 matrix that defines a two-point perspective? What is the form of the matrix that defines a three-point perspective? (Hint: Try multiplying M_{per} by various rotation matrices.)

6.13 What is the effect of applying M_{per} to points whose z coordinate is less than zero?

6.14 Devise a clipping algorithm for a cone-shaped (having a circular cross-section) view volume. The cone's apex is at the origin, its axis is the positive z axis, and it has a 90° interior angle. Consider using a spherical coordinate system.

6.15 Design and implement a set of utility subroutines to generate a 4×4 transformation matrix from an arbitrary sequence of R, S, and T primitive transformations.

6.16 Draw a decision tree to be used when determining the type of a projection used in creating an image. Apply this decision tree to the figures in this chapter that are projections from 3D.

6.17 Evaluate the speed tradeoffs of performing 3D clipping by using the Cohen–Sutherland algorithm versus using the 3D Cyrus–Beck algorithm. First, write an assembly-language version of the program. Count operations executed for each fundamentally different case treated by the algorithms, and weight the cases equally. (For example, in the Cohen–Sutherland algorithm, trivial accept, trivial reject, and calculating 1, 2, 3, or 4 interesections are the fundamentally different cases.) Ignore the time taken by subroutine calls (assume that in-line code is used). Differentiate instructions by the number of cycles they require.

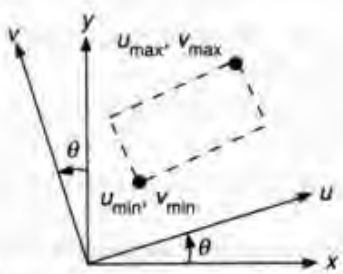


Fig. 6.60 A rotated window.

Samsung/Google, EX1027, p. 299

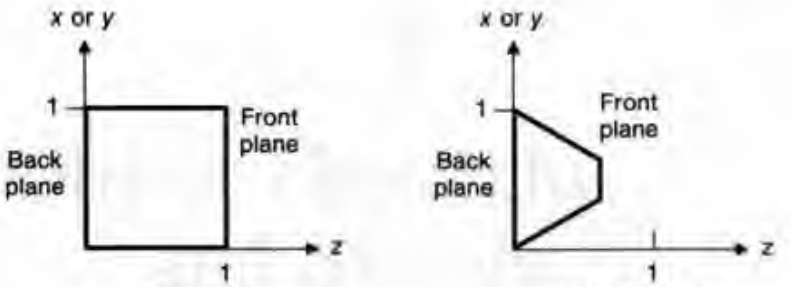


Fig. 6.61 The default view volumes used by PHIGS. The view volumes are slightly different from those used in this chapter, as given in Fig. 6.45.

6.18 In Exercise 6.17, we made the assumption that all cases were equally likely. Is this a good assumption? Explain your answer. Apply either of the algorithms to a variety of real objects, using various view volumes, to get a sense of how the mix of cases varies in a realistic setting.

6.19 The canonical view volume for the parallel projection was taken to be the $2 \times 2 \times 1$ rectangular parallelepiped. Suppose the unit cube in the positive octant, with one corner at the origin, is used instead.

a. Find the normalization N_{par}' for this view volume.
b. Find the corresponding homogeneous-coordinate view volume.

6.20 Give the viewing parameters for top, front, and side views of the house with the VRP in the middle of the window. Must the PRP be different for each of the views? Why or why not?

6.21 Verify that Eq. (6.48) does indeed transform the canonical perspective view volume of Fig. 6.56(a) into the canonical parallel view volume of Fig. 6.56(b).

6.22 In PHIGS, the viewing specification allows the view plane to be at a distance VPD from the VRP. Redevelop the canonical viewing transformations N_{par} and N_{per} to include VPD. Set VPD = 0 to make sure that your results specialize to those in the text.

6.23 Reformulate the derivations of N_{per} and N_{par} to incorporate the matrix $M_{general}$.

6.24 Show how to convert from the six viewing parameters of Fig. 6.35 into the viewing parameters discussed in this chapter. Write and test a utility subroutine to effect the conversion.

6.25 For the homogeneous clip planes $X = -W$, $X = W$, $Y = -W$, $Y = W$, $Z = 0$, $Z = -W$, complete a table like Table 6.1 for using the Cyrus–Beck clipping approach for the $W > 0$ region given by $W > 0$, $-W \leq X \leq W$, $-W \leq Y \leq W$, $-W \leq Z \leq 0$.

6.26 PHIGS uses the default view volumes shown in Fig. 6.61. These are slightly different from the canonical view volumes used in this chapter, as shown in Fig. 6.45. Modify the N_{par} and N_{per} transformations to map into the PHIGS default.

6.27 Stereo pairs are two views of the same scene made from slightly different projection reference points, but with the same view reference point. Let d be the stereo separation; that is, the distance between the two reference points. If we think of the reference points as our eyes, then d is the distance between our eyes. Let P be the point midway between our eyes. Given P, d, VRP, VPN, and VUP, derive expressions for the two projection reference points.

Samsung/Google, EX1027, p. 300

7
Object Hierarchy and Simple PHIGS (SPHIGS)

Andries van Dam
and David Sklar

A graphics package is an intermediary between an application program and the graphics hardware. The output primitives and interaction devices that a graphics package supports can range from rudimentary to extremely rich. In Chapter 2, we described the fairly simple and low-level SRGP package, and we noted some of its limitations. In this chapter, we describe a package based on a considerably richer but more complex standard graphics package, PHIGS (Programmer's Hierarchical Interactive Graphics System[1]). A *standard graphics package* such as PHIGS or GKS (Graphical Kernel System) implements a specification designated as standard by an official national or international standards body; GKS and PHIGS have been so designated by ANSI (American National Standards Institute) and ISO (International Standards Organization). The main purpose of such standards is to promote portability of application programs and of programmers. Nonofficial standards are also developed, promoted, and licensed by individual companies or by consortia of companies and universities; Adobe's PostScript and MIT's X Window System are two of the current industry standards.

The package described here is called SPHIGS (for Simple PHIGS; pronounced "ess-figs") because it is essentially a subset of PHIGS. It preserves most of PHIGS's capabilities and power, but simplifies or modifies various features to suit straightforward applications. SPHIGS also includes several enhancements adapted from PHIGS+ extensions. Our aim in designing SPHIGS has been to introduce concepts in the simplest possible

[1]The term "PHIGS" in this chapter also includes a set of extensions to PHIGS, called PHIGS+, that supports advanced geometric primitives such as polyhedra, curves, and surfaces, as well as rendering techniques that use lighting, shading, and depth-cueing, discussed in Chapters 14–16.

Samsung/Google, EX1027, p. 301

way, not to provide a package that is strictly upward-compatible with PHIGS. However, an SPHIGS application can easily be adapted to run with PHIGS. Footnotes present some of the important differences between SPHIGS and PHIGS; in general, an SPHIGS feature is also present in PHIGS unless otherwise noted.

There are three major differences between SPHIGS and integer raster packages, such as SRGP or the Xlib package of the X Window System. First, to suit engineering and scientific applications, SPHIGS uses a 3D, floating-point coordinate system, and implements the 3D viewing pipeline discussed in Chapter 6.

The second, farther-reaching difference is that SPHIGS maintains a database of structures. A *structure* is a logical grouping of primitives, attributes, and other information. The programmer can modify structures in the database with a few editing commands; SPHIGS ensures that the screen's image is an accurate representation of the contents of the stored database. Structures contain not only specifications of primitives and attributes, but also invocations of subordinate structures. They thus exhibit some of the properties of procedures in programming languages. In particular, just as procedure hierarchy is induced by procedures invoking subprocedures, structure hierarchy is induced by structures invoking substructures. Such hierarchical composition is especially powerful when one can control the geometry (position, orientation, size) and appearance (color, style, thickness, etc.) of any invocation of a substructure.

The third difference is that SPHIGS operates in an abstract, 3D world-coordinate system, not in 2D screen space, and therefore does not support direct pixel manipulation. Because of these differences, SPHIGS and SRGP address different sets of needs and applications; as we pointed out in Chapter 2, each has its place—no single graphics package meets all needs.

Because of its ability to support structure hierarchy, SPHIGS is well suited to applications based on models with component–subcomponent hierarchy; indeed, the SPHIGS structure hierarchy can be viewed as a special-purpose modeling hierarchy. We therefore look at modeling in general in Section 7.1, before discussing the specifics of geometric modeling with SPHIGS. Sections 7.2 through 7.9 show how to create, display, and edit the SPHIGS structure database. Section 7.10 discusses interaction, particularly pick correlation. The remainder of the chapter presents PHIGS features not found in SPHIGS, discusses implementation issues, and closes with evaluations of SPHIGS and of alternative methods for encoding hierarchy.

7.1 GEOMETRIC MODELING

7.1.1 What Is a Model?

You have encountered many examples of models in courses in the physical and social sciences. For example, you are probably familiar with the Bohr model of the atom, in which spheres representing electrons orbit a spherical nucleus containing neutron and proton spheres. Other examples are the exponential unconstrained growth model in biology, and macro- or microeconometric models that purport to describe some aspect of an economy. A model is a representation of some (not necessarily all) features of a concrete or abstract entity. The purpose of a model of an entity is to allow people to visualize and understand

Samsung/Google, EX1027, p. 302

the structure or behavior of the entity, and to provide a convenient vehicle for "experimentation" with and prediction of the effects of inputs or changes to the model. Quantitative models common in physical and social sciences and engineering are usually expressed as systems of equations, and the modeler will experiment by varying the values of independent variables, coefficients, and exponents. Often, models simplify the actual structure or behavior of the modeled entity to make the model easier to visualize or, for those models represented by systems of equations, to make the model computationally tractable.

We restrict ourselves in this book to the discussion of computer-based models—in particular, to those that lend themselves to graphic interpretation. Graphics can be used to create and edit the model, to obtain values for its parameters, and to visualize its behavior and structure. The model and the graphical means for creating and visualizing it are distinct; models such as population models need not have any inherent graphical aspects. Among common types of models for which computer graphics is used are these:

- *Organizational models* are hierarchies representing institutional bureaucracies and taxonomies, such as library classification schemes and biological taxonomies. These models have various directed-graph representations, such as the organization chart.
- *Quantitative models* are equations describing econometric, financial, sociological, demographic, climatic, chemical, physical, and mathematical systems. These are often depicted by graphs or statistical plots.
- *Geometric models* are collections of components with well-defined geometry and, often, interconnections between components, including engineering and architectural structures, molecules and other chemical structures, geographic structures, and vehicles. These models are usually depicted by block diagrams or by pseudorealistic "synthetic photographs."

Computer-assisted modeling allows pharmaceutical drug designers to model the chemical behavior of new compounds that may be effective against particular diseases, aeronautical engineers to predict wing deformation during supersonic flight, pilots to learn to fly, nuclear-reactor experts to predict the effects of various plant malfunctions and to develop the appropriate remedies, and automobile designers to test the integrity of the passenger compartment during crashes. In these and many other instances, it is far easier, cheaper, and safer to experiment with a model than with a real entity. In fact, in many situations, such as training of space-shuttle pilots and studies of nuclear-reactor safety, modeling and simulation are the only feasible method for learning about the system. For these reasons, computer modeling is replacing more traditional techniques, such as wind-tunnel tests. Engineers and scientists now can perform many of their experiments with digital wind tunnels, microscopes, telescopes, and so on. Such numerically based simulation and animation of models is rapidly becoming a new paradigm in science, taking its place beside the traditional branches of theory and physical experimentation. Needless to say, the modeling and simulation are only as good as the model and its inputs—the caution "garbage in, garbage out" pertains especially to modeling.

Models need not necessarily contain intrinsically geometric data; abstractions such as organizational models are not spatially oriented. Nonetheless, most such models can be

Samsung/Google, EX1027, p. 303

represented geometrically; for example, an organizational model may be represented by an organization chart, or the results of a clinical drug evaluation may be represented by a histogram. Even when a model represents an intrinsically geometric object, no fixed graphical representation in the model or view of that model is dictated. For example, we can choose whether to represent a robot as a collection of polyhedra or of curved surfaces, and we can specify how the robot is to be "photographed"—from which viewpoint, with which type of geometric projection, and with what degree of realism. Also, we may choose to show either the structure or the behavior of a model pictorially; for instance, we may want to see both a VLSI circuit's physical layout on the chip and its electrical and logical behaviors as functions of inputs and time.

7.1.2 Geometric Models

Geometric or graphical models describe components with inherent geometrical properties and thus lend themselves naturally to graphical representation. Among the ingredients a geometric model may represent are the following:

- Spatial layout and shape of components (i.e., the *geometry* of the entity), and other attributes affecting the appearance of components, such as color
- Connectivity of components (i.e., the structure or *topology* of the entity); note that connectivity information may be specified abstractly (say, in an adjacency matrix for networks or in a tree structure for a hierarchy), or may have its own intrinsic geometry (the dimensions of channels in an integrated circuit)
- Application-specific data values and properties associated with components, such as electrical characteristics or descriptive text.

Associated with the model may be processing algorithms, such as linear-circuit analysis for discrete circuit models, finite-element analysis for mechanical structures, and energy minimization for molecular models.

There is a tradeoff between what is stored explicitly in the model and what must be computed prior to analysis or display—a classical space–time tradeoff. For example, a model of a computer network could store the connecting lines explicitly or could recompute them from a connectivity matrix with a simple graph-layout algorithm each time a new view is requested. Enough information must be kept with the model to allow analysis and display, but the exact format and the choices of encoding techniques depend on the application and on space-time tradeoffs.

7.1.3 Hierarchy in Geometric Models

Geometric models often have a hierarchical structure induced by a bottom-up construction process: Components are used as building blocks to create higher-level entities, which in turn serve as building blocks for yet higher-level entities, and so on. Like large programming systems, hierarchies are seldom constructed strictly bottom-up or top-down; what matters is the final hierarchy, not the exact construction process. Object hierarchies are common because few entities are monolithic (indivisible); once we decompose an entity into a collection of parts, we have created at least a two-level hierarchy. In the uncommon

Samsung/Google, EX1027, p. 304

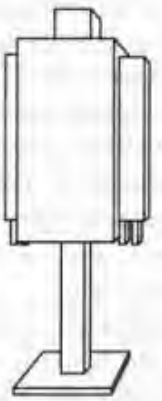


Fig. 7.1 Perspective view of simplified android robot.

case that each object is included only once in a higher-level object, the hierarchy can be symbolized as a tree, with objects as nodes and inclusion relations between objects as edges. In the more common case of objects included multiple times, the hierarchy is symbolized by a *directed acyclic graph* (DAG). As a simple example of object hierarchy, Fig. 7.1 shows a perspective view of a rudimentary "android" robot; Fig. 7.2(a) shows the robot's structure as a DAG. Note that we can duplicate the multiply included objects to convert the DAG to a tree (Fig. 7.2b). By convention, the arrows are left off as redundant because the ordering relationship between nodes is indicated by the nodes' relative positions in the tree—if node A is above node B, then A includes B.

The robot is composed of an upper body swiveling on a base. The upper body is composed of a head that rotates relative to a trunk; the trunk also has attached to it two identical arms that may rotate independently through a horizontal axis "at the shoulder." The arm is composed of a fixed part, "the hand," and a thumb that slides parallel to the hand to form a primitive gripper. Thus, the thumb object is invoked once in the arm, and the arm object is invoked twice in the upper body. We discuss the creation of this robot throughout this chapter; its shape is also presented in three orthographic projections in windows 2–4 of the screen shown in Fig. 7.7(b).

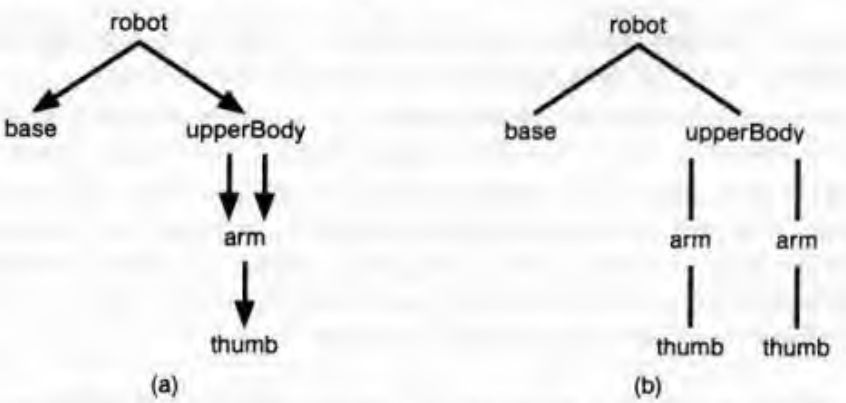


Fig. 7.2 Hierarchy of robot components. (a) Directed acyclic graph (DAG). (b) Tree.

Samsung/Google, EX1027, p. 305

Although an object in a hierarchy is composed of geometric primitives as well as inclusions of lower-level subobjects, the DAG and the tree representing the robot's hierarchy show only references to subobjects. This is analogous to the procedure-hierarchy diagram commonly used to show the calling structure of a program in a high-level procedural language. It is important to note that it is up to the designer to decide exactly how a composite object is hierarchically constructed. For example, the robot could have been modeled as a two-level hierarchy, with a root object consisting of a base, a head and a trunk as geometric primitives (parallelepipeds, say), and two references to an "atomic" arm object composed of geometric primitives only.

Many systems, such as computer networks or chemical plants, can be represented by network diagrams, in which objects are not only included multiple times, but are also interconnected arbitrarily. Such networks are also modeled as graphs, possibly even containing cycles, but they can still exhibit properties of object-inclusion hierarchy when subnetworks occur multiple times.

To simplify the task of building complex objects (and their models), we commonly use application-specific atomic components as the basic building blocks. In 2D, these components are often drawn by using plastic or computer-drawn templates of standard symbolic shapes (also called symbols or stencils). In drawing programs, these shapes, in turn, are composed of geometric primitives, such as lines, rectangles, polygons, ellipse arcs, and so on. In 3D, shapes such as cylinders, parallelepipeds, spheres, pyramids, and surfaces of revolution are used as basic building blocks. These 3D shapes may be defined in terms of lower-level geometric primitives, such as 3D polygons; in this case, smoothly curved surfaces must be approximated by polygonal ones, with attendant loss of resolution. Alternatively, in advanced modeling systems that deal directly with free-form surfaces or volumes, shapes such as parametric polynomial surfaces, and solids such as cylinders, spheres, and cones, are themselves geometric primitives and are defined analytically, without loss of resolution—see Chapters 11 and 12. We use the term *object* in this chapter for those 2D or 3D components that are defined in their own modeling coordinate systems in terms of geometric primitives and lower-level objects, and that usually have not only geometrical data but also associated application data. An object is thus a (composite) shape and all its data.

A hierarchy, then, is created for a variety of purposes:

- To construct complex objects in a modular fashion, typically by repetitive invocations of building blocks that vary in geometric and appearance attributes
- To increase storage economy, since it suffices to store only references to objects that are used repeatedly, rather than the complete object definition each time
- To allow easy update propagation, because a change in the definition of one building-block object is automatically propagated to all higher-level objects that use that object (since they now refer to an updated version); the analogy between object and procedure hierarchy is useful here, in that a change to the body of a procedure is also reflected in all invocations of that procedure.

The application can use a variety of techniques to encode hierarchical models. For example, a network or relational database can be used to store information on objects and

Samsung/Google, EX1027, p. 306

on relationships between objects. Alternatively, a more efficient, customized linked-list structure can be maintained by the application program, with records for objects and pointers for relationships. In some models, connections between objects are objects in their own right; they must also be represented with data records in the model. Yet another alternative is to use an object-oriented database [ZDON90]. Object-oriented programming environments such as SmallTalk [GOLD83], MacApp [SCHM86] and ET++ [WEIN88] are increasingly being used to store modeling information for the geometric objects in graphics application programs.

Interconnections. In most networks, objects are placed in specified locations (either interactively by the user or automatically by the application program) and then are interconnected. Interconnections may be abstract and thus of arbitrary shape (e.g., in hierarchy or network diagrams, such as organization charts or project-scheduling charts), or they may have significant geometry of their own (e.g., a VLSI chip). If connections are abstract, we can use various standard conventions for laying out hierarchical or network diagrams, and we can employ attributes such as line style, line width, or color to denote various types of relationships (e.g., "dotted-line responsibility" in an organization chart). Connections whose shape matters, such as the channels connecting transistors and other components of a VLSI circuit, are essentially objects in their own right. Both abstract and nonabstract connections are often *constrained* to have horizontal or vertical orientations (sometimes called the *Manhattan* layout scheme) to simplify visualization and physical construction.

Parameter passing in object hierarchy. Objects invoked as building blocks must be positioned in exactly the right place in their parent objects, and, in order to fit, often must be resized and reoriented as well. Homogeneous coordinate matrices were used to transform primitives in Chapter 5 and to normalize the view volume in Chapter 6, and it should come as no surprise that, in a hierarchical model, one frequently applies scaling, rotation, and translation matrices to subobjects. Sutherland first used this capability for graphical modeling in Sketchpad [SUTH63], coining the terms *master* for the definition of an object and *instance* for a geometrically transformed invocation. As discussed in Section 4.3.3, graphics systems using hierarchical display lists (also called structured display files) implement master–instance hierarchy in hardware, using subroutine jumps and high-speed floating-point arithmetic units for transformations. Because we want to distinguish geometric transformations used in normalizing the view volume from those used in building object hierarchy, we often speak of the latter as *modeling transformations*. Mathematically, of course, there is no difference between modeling and normalizing transformations.

Once again, in analogy with procedure hierarchy, we sometimes speak of a parent object "calling" a child object in a hierarchy, and passing it "geometric parameters" corresponding to its scale, orientation, and position in the parent's coordinate system. As we see shortly, graphics packages that support object hierarchy, such as SPHIGS, can store, compose, and apply transformation matrices to vertices of primitives, as well as to vertices of instantiated child objects. Furthermore, attributes affecting appearance can also be passed to instantiated objects. In Section 7.5.3, however, we shall see that the SPHIGS parameter-passing mechanism is not as general as is that of a procedural language.

Samsung/Google, EX1027, p. 307

7.1.4 Relationship between Model, Application Program, and Graphics System

So far, we have looked at models in general, and geometric models with hierarchy and modeling transformations in particular. Before looking at SPHIGS, let us briefly review the conceptual model of graphics first shown in Fig. 1.5 and elaborated in Fig. 3.2, to show the interrelationship between the model, the application program, and the graphics system. In the diagram in Fig. 7.3, application programs are divided into five subsystems, labeled (a) through (e):

a. Build, modify, and maintain the model by adding, deleting, and replacing information in it

b. Traverse (scan) the model to extract information for display

c. Traverse the model to extract information used in the analysis of the model's behavior/performance

d. Display both information (e.g., rendering of a geometric model, output of an analysis) and user-interface "tools" (e.g., menus, dialog boxes)

e. Perform miscellaneous application tasks not directly involving the model or display (e.g., housekeeping).

The term *subsystem* does not imply major modules of code—a few calls or a short procedure may be sufficient to implement a given subsystem. Furthermore, a subsystem may be distributed throughout the application program, rather than being gathered in a separate program module. Thus, Fig. 7.3 simply shows logical components, not necessarily program structure components; moreover, while it does differentiate the procedures that

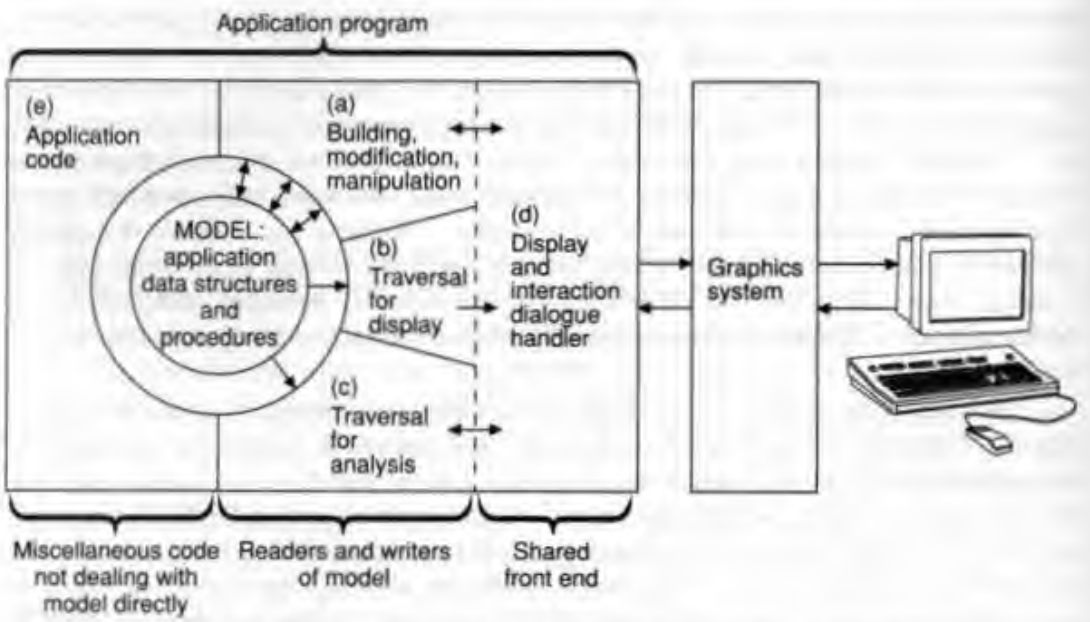


Fig. 7.3 The application model and its readers and writers.

Samsung/Google, EX1027, p. 308

build, modify, analyze, or display the model, it is not always clear whether to call a particular module part of the model or part of the model-maintenance code. It could be argued, for example, that a circuit-analysis module is really part of the model's definition because it describes how the circuit behaves. For a programmer using a traditional procedural language such as Pascal or C, Fig. 7.3 works best if one thinks of the model as primarily containing data. People familiar with object-oriented programming languages will find the mixture of data and procedures a natural one.

In many application programs, especially industrial ones, an "80/20" rule holds: the major portion of the program (80 percent) deals with modeling of entities, and only a minor portion (20 percent) deals with making pictures of them. In other words, in many applications such as CAD, pictorial representation of models is a means to an end, the end being analysis, physical construction, numerically controlled machining, or some other type of postprocessing. Naturally, there are also many applications for which "the picture is the thing"—for example, painting, drafting, film and video production, and animation of scenes for flight simulators. Of these, all but painting also require a model from which the images are rendered. In short, most graphics involves significant modeling (and simulation) and there is considerable support for the saying, "graphics *is* modeling"; Chapters 11, 12, and 20 are devoted to that important subject.

7.2 CHARACTERISTICS OF RETAINED-MODE GRAPHICS PACKAGES

In discussing the roles of the application program, the application model, and the graphics package, we glossed over exactly what capabilities the graphics package has and what happens when the model is modified. As noted in Chapter 2, SRGP operates in *immediate mode* and keeps no record of the primitives and attributes passed to it. Thus, deletions of and changes to application objects necessitate the removal of information on the screen and therefore either selective modification or complete regeneration of the screen; either of these requires the application to respecify primitives from its model. PHIGS, on the other hand, operates in *retained mode*: It keeps a record of all primitives and other related information to allow subsequent editing and automatic updating of the display, thereby offloading the application program.

7.2.1 Central Structure Storage and its Advantages

PHIGS stores information in a special-purpose database called the *central structure storage* (CSS). A *structure* in PHIGS is a sequence of *elements*—primitives, appearance attributes, transformation matrices, and invocations of subordinate structures—whose purpose is to define a coherent geometric object. Thus, PHIGS effectively stores a special-purpose modeling hierarchy, complete with modeling transformations and other attributes passed as "parameters" to subordinate structures. Notice the similarities between the CSS modeling hierarchy and a hardware hierarchical display list that stores a master–instance hierarchy. In effect, PHIGS may be viewed as the specification of a device-independent hierarchical display-list package; a given implementation is, of course, optimized for a particular display device, but the application programmer need not be concerned with that. Whereas many

Samsung/Google, EX1027, p. 309

PHIGS implementations are purely software, there is increasing use of hardware to implement part or all of the package.

As does a display list, the CSS duplicates geometric information stored in the application's more general-purpose model/database to facilitate rapid *display traversal*—that is, the traversal used to compute a new view of the model. The primary advantage of the CSS, therefore, is rapid automatic screen regeneration whenever the application updates the CSS. This feature alone may be worth the duplication of geometric data in the application database and the CSS, especially when the PHIGS implementation uses a separate processor as a "traversal engine" to offload display traversal from the CPU running the application (see Chapter 18). Small edits, such as changing a transformation matrix, are also done efficiently in PHIGS.

A second advantage of the CSS is automatic pick correlation: The package determines the identity and place within the hierarchy of the primitive picked by the user (see Sections 7.10.2 and 7.12.2). The pick-correlation facility exemplifies a common technique of moving frequently needed functionality into the underlying graphics package.

A third advantage of the CSS is that its editing facilities, in combination with the features of hierarchical modeling, make it easy to create various dynamic effects—for example, motion dynamics—in which time-varying transformations are used to scale, rotate, and position subobjects within parent objects. For example, we can model our simple robot so that each joint is represented by a rotation applied to a substructure (e.g., the arm is a rotated subordinate of the upper body), and dynamically rotate the arm by editing a single rotation matrix.

7.2.2 Limitations of Retained-Mode Packages

Although the CSS (as a special-purpose entity built primarily for display and fast incremental updating) facilitates certain common modeling operations, it is neither necessary nor sufficient for all modeling purposes. It is not necessary because an application can do its own screen regeneration when the model is changed, can do its own pick correlation (albeit with considerable work), and can implement its own object hierarchy via procedures defining objects and accepting transformations and other parameters. The CSS is generally not sufficient because, in most applications, a separately built and updated application data structure is still necessary to record all appropriate data for each application object. Thus, there is duplication of all geometric data, and the two representations must be synchronized properly. For all these reasons, some graphics packages support floating-point coordinates and generalized 2D and 3D viewing facilities without any type of structure storage. The rationale for such immediate-mode packages is that maintaining the CSS is often not worth the overhead, since the application typically maintains an application model sufficient for regenerating the screen image.

For applications in which there is significant structural change between successive images, using a retained-mode package does not pay. For example, in a "digital–wind-tunnel" analysis of an airplane wing, where the surface is represented by a mesh of triangles, most of the vertices shift slightly in position as the wing is subjected to aerodynamic forces. Editing a structure database for such a case makes no sense, since most of the data are replaced for each new image. Indeed, editing the PHIGS structure database is not advised unless the number of elements to be edited is small relative to the size of the

Samsung/Google, EX1027, p. 310

networks being displayed. The editing tools provided by PHIGS are rudimentary; for example, it is easy to change a modeling transformation, but to change a vertex of a polygon requires deleting the polygon and respecifying the changed version. Typically, implementations are likely to be optimized for display traversal, since that is the most common operation, rather than for massive editing. Furthermore, the application model must be updated in any case, and it is easier and faster to update just one database than to update two of them.

Because of these limitations, some implementations of PHIGS offer an immediate-mode output facility, although for technical reasons that facility is not part of the official PHIGS specification.

7.3 DEFINING AND DISPLAYING STRUCTURES

The previous section has discussed general properties of PHIGS and SPHIGS. With this section, we begin describing the SPHIGS package in detail; unless otherwise noted, the discussion is generally also applicable to PHIGS. The manipulations permitted on SPHIGS structures include the following:

- Opening (to initiate editing) and closing (to conclude editing)
- Deleting
- Inserting *structure elements* (the three primary types of structure elements are primitives, attributes, including those that specify modeling transformations, and elements that invoke substructures). An element is a data record that is created and inserted into the currently open structure whenever an *element-generator* procedure is called and that stores that procedure's parameters.
- Deleting structure elements
- *Posting* for display (by analogy to posting a snapshot on a bulletin board), subject to a *viewing operation* that specifies how to map the floating-point coordinates to the screen's coordinate system.

7.3.1 Opening and Closing Structures

To create a structure—for example, the collection of primitives and attributes forming the arm component of the robot in Fig. 7.2—we bracket calls to the element-generator procedures with calls to

```
void SPH_openStructure (int structureID);
void SPH_closeStructure (void);
```

These procedures do for structures essentially what the standard open- and close-file commands do for disk files. Unlike disk files, however, only one structure may be open at any time, and all elements specified while it is open are stored in it. Once closed, structures may be reopened for editing (see Section 7.9).

We note two additional properties of structures. First, primitives and attributes can be specified only as elements of a structure (much as all statements in a C program must be specified in some procedure or function). There are no rules about how many elements may be stored in a structure; a structure can be empty, or can contain an arbitrarily large

Samsung/Google, EX1027, p. 311

number of elements, limited only by memory space. Of course, the elements forming a structure should be, in general, a logically cohesive set defining a single object.

Second, structure IDs are integers. Since they are normally used only by the application program, not by the interactive user, they do not need to have the more general form of character strings, although the application programmer is free to define symbolic constants for structure IDs. Integer IDs also allow a convenient mapping between objects in the application data structure and the objects' corresponding structure IDs.

7.3.2 Specifying Output Primitives and Their Attributes

The procedures that generate output-primitive elements look like their SRGP counterparts, but there are important differences. First, points are specified with three double-precision coordinates (x, y, and z). Moreover, these procedures place elements in the currently open structure in the CSS rather than directly altering the screen image—displaying structures is a separate operation described in Section 7.3.3. In this chapter, the term *primitive* is used as shorthand for three related entities: an element-generation procedure, such as SPH_polyLine; the structure element generated by that procedure (for example, the polyLine element); and the displayed image created when a primitive element is executed during display traversal of central structure storage. SPHIGS *executes* a primitive element by transforming the primitive's coordinates by modeling transformations and a viewing operation, including clipping it to the view volume, and then *rasterizing* it (i.e., converting it to pixels). Attributes are more specialized than in SRGP, in that each type of primitive has its own attributes. Thus, attributes such as color and line style are in effect "typed," so that the programmer can, for example, reset the current color for lines while preserving the current colors for polyhedra and text.

Primitives. SPHIGS has fewer output primitives than SRGP does, because the 3D "solid" equivalents of some of SRGP's primitives (e.g., an ellipsoid) are computationally expensive to implement, especially with respect to transformations, clipping, and scan conversion.

Most of the SPHIGS primitives are identical to their SRGP counterparts in their specification methods (except that the points are 3D):

```
void SPH_polyLine (int vertexCount, point *vertices);
void SPH_polyMarker (int vertexCount, point *vertices);
void SPH_fillArea (int vertexCount, point *vertices);     /* Like SRGP_polygon */
void SPH_text (point origin, char *str);                  /* Not fully 3D; see Section 7.7.2 */
```

Note that SPHIGS does not verify that fill areas (or facets, described next) are planar, and the results are undefined if they are not.

Now, consider the definition of a simple house, shown in Fig. 7.4. We can describe this house to SPHIGS by specifying each face (also called a *facet*) as a fill area, at the cost of unnecessary duplication in the specification and storage (in CSS) of each shared vertex. This duplication also slows down display generation, since the viewing-operation calculation would be performed on a single vertex more than once. It is far more efficient in storage

Samsung/Google, EX1027, p. 312

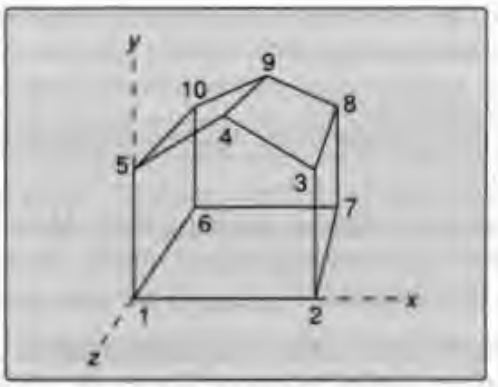


Fig. 7.4 A simple house defined as a set of vertices and facets.

and processing time to specify the facets using indirect references to the shared vertices. We thus think of a polyhedron as a collection of facets, each facet being a list of vertex indices and each index being a "pointer" into a list of vertices. We can describe a polyhedron's specification using the following notation:

Polyhedron = {VertexList, FacetList}
VertexList = {V1, V2, V3, V4, V5, V6, V7, V8, V9, V10}
V1 = (x_1, y_1, z_1), V2 = (x_2, y_2, z_2) . . .
FacetList = {front = {1, 2, 3, 4, 5}, right = {2, 7, 8, 3}, . . . bottom = {. . .} }

SPHIGS offers this efficient form of specification with its *polyhedron* primitive. In SPHIGS terminology, a polyhedron is a collection of facets that may or may not enclose a volume. In a closed polyhedron such as our house, vertices are typically shared by at least three facets, so the efficiency of the indirect method of specification is especially high. The appearance of a polyhedron is affected by the same attributes that apply to fill areas.

The list of facets is presented to the polyhedron element generator in the form of a single array of integers (SPHIGS type "vertexIndex*") storing a concatenated set of facet descriptions. Each facet description is a sequence of $(V+1)$ integers, where V is the number of vertices in the facet. The first V integers are indices into the vertex list; the last integer is (-1) and acts as a sentinel ending the facet specification. Thus, we would specify the facets of the house (via the fourth parameter of the procedure, described next) by sending the array: 1, 2, 3, 4, 5, −1, 2, 7, 8, 3, −1, ...

void SPH_polyhedron (**int** *vertexCount*, **int** *facetCount*, point **vertices*, vertexIndex **facets*);

Note that the SPHIGS rendering algorithms require that a facet's two sides be distinguishable (external versus internal). Thus, the vertices must be specified in counterclockwise (right-hand rule) order, as one examines the external side of the facet.[2]

[2] SPHIGS requires that one side of each facet be deemed "external," even if the polyhedron's facets do not form a closed object. Furthermore, the "internal" side of a polyhedron facet is never visible.

Samsung/Google, EX1027, p. 313

As a simple example, the following C code creates a structure consisting of a single polyhedron modeling the house of Fig. 7.4:

```
SPH_openStructure (HOUSE_STRUCT);
    SPH_polyhedron (10, 7, houseVertexList, houseFacetDescription);
SPH_closeStructure();
```

In essence, SPHIGS supports polygonal geometry only. More advanced 3D modeling primitives are covered later — polynomially defined smooth curves and surfaces in Chapter 11, and solid primitives in Chapter 12.

Attributes. The procedures listed in Fig. 7.5 generate attribute elements. During display traversal, the execution of an attribute element changes the attribute's value in a modal fashion: The new value stays in effect until it is changed explicitly. Attributes are bound to primitives during display traversal, as discussed in the next section and in Section 7.7.

The attributes of fill areas are different from those in SRGP. Both fill-area and polyhedron primitives have interiors and edges whose attributes are specified separately. The interior has only the color attribute, whereas the edge has style, width, and color attributes. Moreover, the visibility of edges can be turned off via the edge-flag attribute, which is useful for various rendering modes, as discussed in Section 7.8.

Line/edge width and marker size are specified in a *nongeometric* manner: They are not defined using world-coordinate–system units and therefore are not subject to geometric transformations. Thus, modeling transformations and the viewing operation may change a line's apparent length, but not its width. Similarly, the length of dashes in a noncontinuous line style is independent of the transformations applied to the line. However, unlike in SRGP, pixels are not used as the units of measurement, because their sizes are device-dependent. Rather, a nominal width/size has been preset for each device, so that a

polyLine:

```
void SPH_setLineStyle (CONTINUOUS / DASHED / DOTTED / DOT_DASHED lineStyle);
void SPH_setLineWidthScaleFactor (double scaleFactor);
void SPH_setLineColor (int colorIndex);
```

fill area and polyhedron:

```
void SPH_setInteriorColor (int colorIndex);
void SPH_setEdgeFlag (EDGE_VISIBLE / EDGE_INVISIBLE flag);
void SPH_setEdgeStyle (CONTINUOUS / DASHED / DOTTED / DOT_DASHED lineStyle);
void SPH_setEdgeWidthScaleFactor (double scaleFactor);
void SPH_setEdgeColor (int colorIndex);
```

polyMarker:

```
void SPH_setMarkerStyle (MARKER_CIRCLE / MARKER_SQUARE/... markerStyle);
void SPH_setMarkerSizeScaleFactor (double scaleFactor);
void SPH_setMarkerColor (int colorIndex);
```

text:

```
void SPH_setTextFont (int fontIndex);
void SPH_setTextColor (int colorIndex);
```

Fig. 7.5 Procedures generating attribute elements.

Samsung/Google, EX1027, p. 314

unit of width/size will have roughly the same appearance on any output device; the SPHIGS application specifies a (noninteger) multiple of that nominal width/size.

SPHIGS does not support patterns, for three reasons. First, SPHIGS reserves patterns to simulate color shading on bilevel systems. Second, smooth shading of patterned areas on a color system is much too computationally intensive for most display systems. Third, the type of geometric pattern called "hatching" in PHIGS is also too time-consuming, even for display systems with real-time transformation hardware.

7.3.3 Posting Structures for Display Traversal

SPHIGS records a newly created structure in the CSS, but does not display it until the application *posts* the structure subject to a particular viewing specification.[3] SPHIGS then performs a *display traversal* of the structure's elements in the CSS, executing each element in order from the first to the last. Executing a primitive element contributes to the screen image (if a portion of the primitive is in view). Executing an attribute element (both geometric transformations and appearance attributes) changes the collection of attributes stored in a state vector (the *attribute-traversal state*) that is applied to subsequent primitives as they are encountered, in modal fashion. Thus, attributes are applied to primitives in display-traversal order.

The following procedure adds a structure to the list of posted structures maintained internally by SPHIGS:

void SPH_postRoot (**int** *structureID*, **int** *viewIndex*);

The term *root* indicates that, in posting a structure S that invokes substructures, we are actually posting the hierarchical DAG, called the *structure network*, whose root is S. Even if a posted structure does not invoke substructures, it is called a root; all posted structures are roots.

The *viewIndex* parameter chooses an entry in the table of *views* (discussed in the next section); this entry specifies how the coordinates of the structure's primitives are to be mapped to the screen's integer coordinate space.

We can erase an object's image from the screen by deleting structures (or elements) from the CSS (see Section 7.9) or by using the less drastic SPH_unpostRoot procedure that removes the root from the list of posted roots, without deleting it from the CSS:

void SPH_unpostRoot (**int** *structureID*, **int** *viewIndex*);

7.3.4 Viewing

The synthetic camera. It is helpful to think of a 3D graphics package as a synthetic camera that takes "snapshots" of a 3D world inhabited by geometrically defined objects. Creating a structure is equivalent to positioning an object in a photography studio; posting a structure is analogous to activating an instant camera previously set up to point at the scene, and then having the snapshot of the scene posted on a bulletin board. As we see shortly,

[3] This way of specifying structure display is the most significant difference between PHIGS and SPHIGS. In PHIGS's more general mechanism, the view specification is a structure element; this allows the view to be changed during display traversal and to be edited like any other element. Many current PHIGS implementations also support the simpler SPHIGS-style posting mechanism.

Samsung/Google, EX1027, p. 315

each time anything changes in the scene, our synthetic camera automatically produces a new, updated image that is posted in place of the old one. To create animation, we show multiple static images in rapid succession, as a movie camera does.

Continuing the metaphor, let us consider how the synthetic picture is produced. First, the camera operator must position and orient the camera; then, he must decide how much of the scene should appear: For example, is the image to be a closeup of a portion of an object of interest, or a long-distance view of the entire scene? Subsequently, the photographer must decide how large a print to make for posting on the bulletin board: Is it to be a wallet-sized print or a poster? Finally, the place on the bulletin board where the photograph is to be posted must be determined. In SPHIGS, these criteria are represented in a *view* that includes a specification of a viewing operation; this operation's *viewport* specifies the size of the photograph and its position on the bulletin board. Not all objects in the structure database need be photographed with the same "camera setting." Indeed, multiple views may be specified for the bulletin board, as we shall see shortly.

The viewport. As discussed in the previous chapter, the viewport specifies a parallelepiped in the NPC system to which the contents of the view volume defined in VRC is mapped. Since the NPC system is mapped to the physical device's integer-coordinate system in a fixed manner, the viewport also specifies where the image is to appear on the screen. The 3D NPC system is mapped to the 2D screen coordinate system in this manner: The NPC unit cube having one corner at (0, 0, 0) and the opposing corner at (1, 1, 1) is mapped to the largest square that can be inscribed on the screen, with the z coordinate simply ignored. For example, on a display device having a resolution of 1024 horizontally and 800 vertically, a point $(0.5, 0.75, z)_{NPC}$ is mapped to $(512, 599)_{DC}$. For portability, an application should not use NPC space lying outside the unit cube; often, however, the benefits of taking full advantage of a nonsquare screen shape are worth the portability cost.

The view table. SPHIGS maintains a table of views that has an implementation-dependent number of entries. Each view consists of a specification of the view volume and viewport, called the *view representation*, and a list (initially empty) of the roots posted to it. Entry 0 in the table defines a *default view* having the volume described in Fig. 6.25 (b), with the front and back planes at $z = 0$ and $z = -\infty$, respectively. The viewport for this default view is the NPC unit cube.

The view representations for all entries in the table (except view 0) may be edited by the application via

```
void SPH_setViewRepresentation (
    int viewIndex, matrix_4x4 voMatrix, matrix_4x4 vmMatrix,
    double NPCviewport_minX, double NPCviewport_maxX,
    double NPCviewport_minY, double NPCviewport_maxY,
    double NPCviewport_minZ, double NPCviewport_maxZ);
```

The two 4 × 4 homogeneous-coordinate matrices are the view-orientation and view-mapping matrices described in Chapter 6. They are produced by the procedures shown in Fig. 7.6.

Samsung/Google, EX1027, p. 316

```
/* To set up UVN viewing reference coordinate system */
matrix_4x4 SPH_evaluateViewOrientationMatrix (
        point viewRefPoint,
        vector_3D viewPlaneNormal, vector_3D viewUpVector,
        matrix_4x4 voMatrix);

/* To set up view volume and to describe how it is to be mapped to NPC space */
matrix_4x4 SPH_evaluateViewMappingMatrix (
        /* First, we specify the view-volume in VRC */
        double umin, double umax, double vmin, double vmax,     /* View-plane boundaries */
        PARALLEL / PERSPECTIVE projectionType,
        point projectionReferencePoint;                          /* In VRC */
        double frontPlaneDistance, double backPlaneDistance,     /* Clipping planes */
        /* Then, we specify the NPC viewport. */
        double NPCvp_minX, double NPCvp_maxX,
        double NPCvp_minY, double NPCvp_maxY,
        double NPCvp_minZ, double NPCvp_maxZ,
        matrix_4x4 vmMatrix);
```

Fig. 7.6 Utilities for calculating viewing-transformation matrices.

Multiple views. The view index specified during posting refers to a specific NPC viewport describing where on the screen (bulletin board) the image of the structure (photograph) is to appear. Just as one can tack several photographic prints on a board, an application can divide the screen into a number of viewports.

The use of multiple views is powerful in many ways. We can display several different structures simultaneously in individual areas of the screen by posting them with different views. In Fig. 7.7(a), we present a schematic representation of the view table, showing only the pointers to the lists of structure networks posted to each view. We can see that there is one view showing a street scene; also, there are three separate views of a robot. The robot

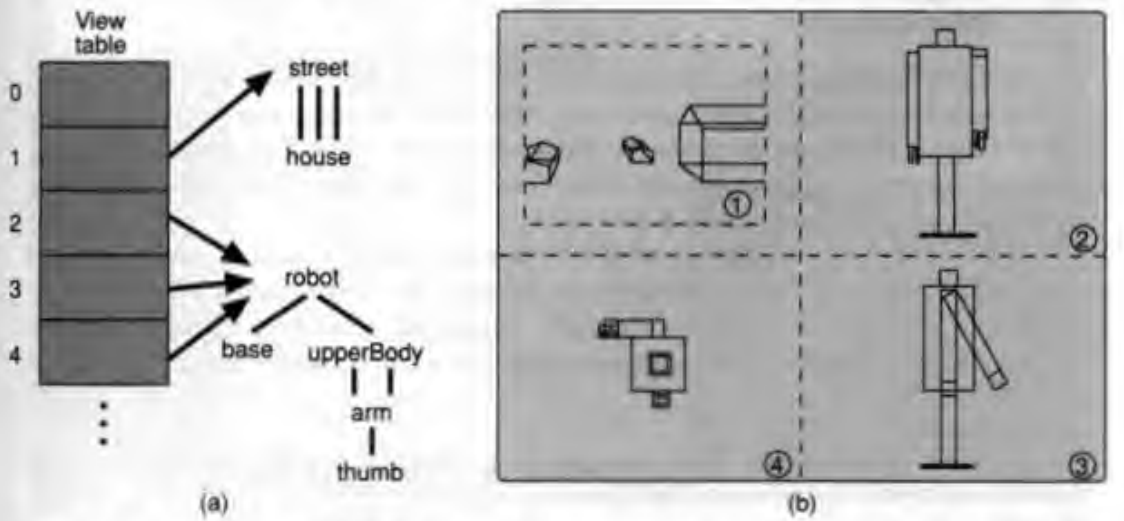


Fig. 7.7 Multiple views sharing screen space. (a) Schematic diagram of view table. Each view entry points to a list of the roots posted to that view. (b) Resulting image. Dashed viewport extents and circled numbers show the viewports and their associated view indices.

Samsung/Google, EX1027, p. 317

structure was posted three times, each time with a different view index. Figure 7.7(b) shows the resulting screen image. The multiple views of the robot vary not only in their viewport specifications, but also in their view-volume specifications.

The preceding scenario implies that each view has at most one posted structure; in fact, however, any number of roots can be posted to a single view. Thus, we can display different root structures in a single unified picture by posting them together to a view. In this case, our metaphorical camera would take a single composite snapshot of a scene that contains many different objects.

Another property of viewports is that, unlike real snapshots and window-manager windows, they are transparent.[4] In practice, many applications *tile* the viewports to avoid overlapping; however, we can also use overlapping to advantage. For example, we can "compose" two distinct images created by different viewing transformations or showing objects defined in different units of measurement; thus, in building a close-up diagram of an engine part, we could inset a small picture showing the entire engine (for context) overlapping the large close-up picture. (To avoid confusion, we would do this by selecting an area of the closeup that is just background.)

To regenerate the screen, SPHIGS displays the posted networks by traversing the roots posted to each view in the view table, in increasing order of view index, starting with view 0; thus, the images of objects posted to view N have *display priority* over the images of objects posted to a view with an index less than N, and therefore potentially overlap them. This ordering is significant, of course, only when viewports actually overlap.[5]

Note that an application can manufacture many independent WC spaces and can use any units of measurement desired. In Fig. 7.7, for example, the street structure is defined in a WC space in which each increment on an axis represents 10 yards, whereas the robot is defined in a wholly independent WC space measured in centimeters. Although each root structure is modeled in its own WC space, there is only one NPC space per display device, shared by all posted structures, since that space is an abstraction of the display device.

7.3.5 Graphics Applications Sharing a Screen via Window Management

When the first standard graphics packages were being designed in the early 1970s, only a single graphics application ran at a given time, and it used the entire screen. The design of PHIGS began in the late 1970s, when this mode of operation was still dominant and before window managers were generally available. Thus, the unit cube of the NPC space was traditionally mapped to the screen in its entirety.

Modern graphics workstations with multitasking operating systems allow multiple graphics applications to run simultaneously, sharing the workstation's resources, the screen, and the set of input devices, under control of a window manager. In this environment, each application is assigned to its own window, which acts as a "virtual

---

[4] Some versions of PHIGS offer opaque viewports as an alternative to tiling; the term *shielding* is typically used to refer to this nonstandard feature.

[5] This trivial view-priority system is less sophisticated than that of PHIGS, which allows explicit view priorities to be assigned by the application.

Samsung/Google, EX1027, p. 318

screen.'' The user can resize and move these windows by calling on the functionality of the window manager. The primary advantage is that each application can act as though it controls an entire screen; it does not need to know that its screen is only a portion of the actual display device's screen. An SPHIGS application therefore need not be modified for a window-manager environment; the package and the window manager cooperate to map NPC space to an assigned window rather than to the entire screen.[6] Figure 7.8 shows two SPHIGS applications and a terminal-emulator program running concurrently on a graphics workstation. Because SPHIGS maps the NPC space to the largest square that fits within the window-manager window, some portion of any nonsquare window is unavailable to the SPHIGS application, as illustrated by the SPHIGS window showing the table and chair scene.

[6] It is sometimes desirable to display only a portion of NPC space in a window-manager window. PHIGS supports the specification of an NPC *workstation window*, which is used to clip the image produced by display traversal; the clipped portion is then mapped to a *workstation viewport* of the same aspect ratio, specified in physical device coordinates. This workstation transformation can also be used to map a rectangular portion of NPC space to the physical display, allowing use of a nonsquare screen region.

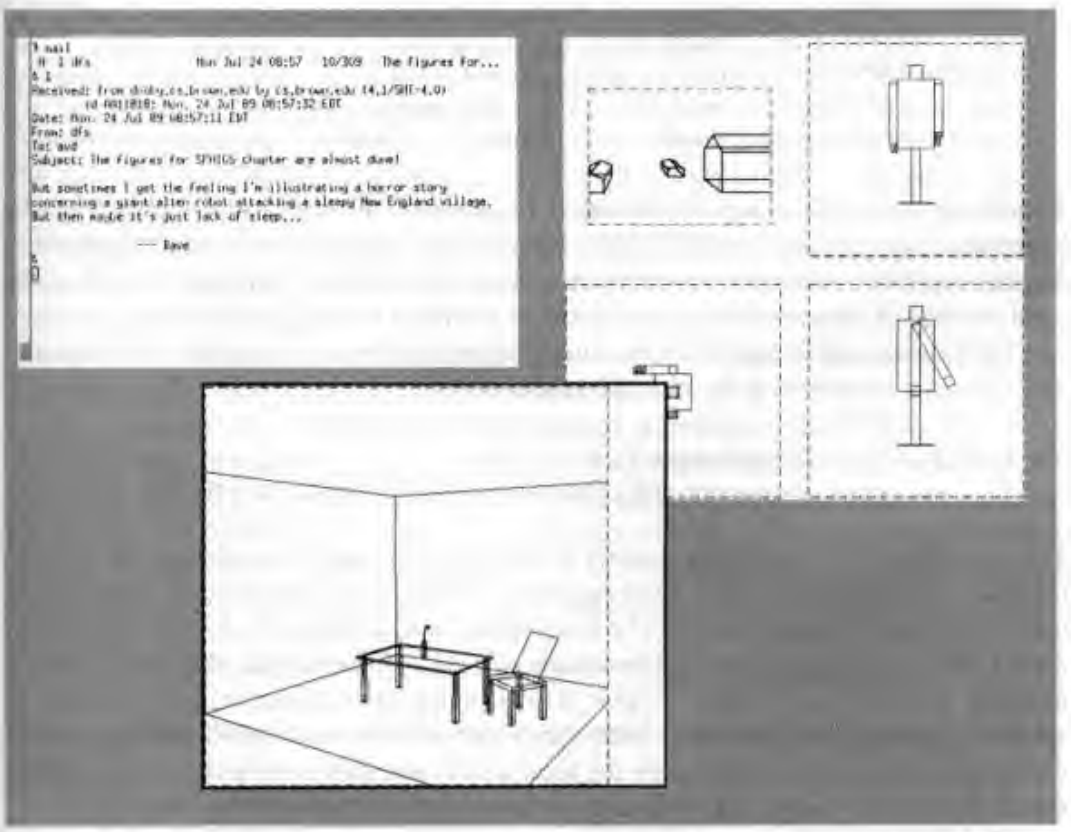


Fig. 7.8 Two SPHIGS applications running in window-manager windows.

Samsung/Google, EX1027, p. 319

7.4 MODELING TRANSFORMATIONS

Section 7.3.2 contained a Pascal code fragment that created a simple structure modeling a house. For simplicity's sake, we placed one of the house's corners at the origin, aligned the house's sides with the principal axes, and gave it dimensions that were whole units. We shall say that an object defined at the origin and (largely) aligned with the principal axes is *standardized*; not only is it easier to define (determine the vertex coordinates of) a standardized object than it is to define one arbitrarily positioned in space, but also it is easier to manipulate the geometry of a standardized object in order to resize, reorient, or reposition it.

Let us say we want our house to appear at a different location, not near the origin. We could certainly recompute the house's vertices ourselves, and create the house structure using the same Pascal code shown in Section 7.3.2 (changing only the vertex coordinates). Instead, however, let us examine the powerful technique of transforming a standardized building-block object in order to change its dimensions or placement.

As we saw in Chapter 5, we can transform a primitive such as a polygon by multiplying each vertex, represented as a column vector $[x, y, z, 1]^T$, by a 4×4 homogeneous-coordinate transformation matrix. The following utility functions create such matrices:

```
matrix_4x4 SPH_scale (double scaleX, double scaleY, double scaleZ, matrix_4x4 result);
matrix_4x4 SPH_rotateX (double angle, matrix_4x4 result);
matrix_4x4 SPH_rotateY (double angle, matrix_4x4 result);
matrix_4x4 SPH_rotateZ (double angle, matrix_4x4 result);
matrix_4x4 SPH_translate (double deltaX, double deltaY, double deltaZ, matrix_4x4 result);
```

A different scale factor may be specified for each of the axes, so an object can be "stretched" or "shrunk" nonuniformly. For rotation, the angle parameter, expressed in degrees, represents counterclockwise motion about the designated principal axis, from the point of view of someone looking along the axis from $+\infty$ toward the origin.

The matrices can be used to create a transformation element to be placed in a structure. The following procedure is the element generator:

```
void SPH_setLocalTransformation (
        matrix_4x4 matrix, REPLACE / PRECONCATENATE / POSTCONCATENATE mode);
```

The use of the prefix "local" here refers to how SPHIGS displays a structure. As SPHIGS traverses a structure, it stores a *local matrix* as a piece of state information applicable to only the structure being traversed. The local matrix is by default initialized to the identity matrix. Whenever a setLocalTransformation element is encountered, the local matrix is modified in some way: It either is replaced or is changed by a multiplication operation, as specified by the *mode* parameter. Whenever a primitive is encountered during traversal, each of its vertices is transformed by the local matrix and then is subjected to the viewing transformation for display. (As we shall see later, hierarchy complicates this rule.)

The following code creates a structure containing our house at an arbitrary location,

Samsung/Google, EX1027, p. 320

and posts that structure for display using the default view. The house maintains its original standardized size and orientation.

```
SPH_openStructure (HOUSE_STRUCT);
    SPH_setLocalTransformation (SPH_translate (...), REPLACE);
    SPH_polyhedron (...);      /* Vertices here are standardized as before */
SPH_closeStructure ();
SPH_postRoot (HOUSE_STRUCT, 0);
```

Simple transformations like this one are uncommon. We typically wish not only to translate the object, but also to affect its size and orientation. When multiple transformations of primitives are desired, the application "builds" the local matrix by successively concatenating (i.e., composing) individual transformation matrices in the exact order in which they are to be applied. In general, standardized building-block objects are scaled, then rotated, and finally translated to their desired location; as we saw in Chapter 5, this order avoids unwanted translations or shearing.

The following code creates and posts a house structure that is moved away from the origin and is rotated to a position where we see its side instead of its front:

```
SPH_openStructure (MOVED_HOUSE_STRUCT);
    SPH_setLocalTransformation (SPH_rotateY (...), REPLACE);
    SPH_setLocalTransformation (SPH_translate (...), PRECONCATENATE);
    SPH_polyhedron (...);      /* Vertices here are standardized as before */
SPH_closeStructure ();
SPH_postRoot (MOVED_HOUSE_STRUCT, 0);
```

The use of the PRECONCATENATE mode for the translation matrix ensures that premultiplication is used to compose the translation matrix with the rotation matrix, and thus that the translation's effect "follows" the rotation's. Premultiplication is thus a far more common mode than is postmultiplication, since it corresponds to the order of the individual transformation elements. Since SPHIGS performs scaling and rotation relative to the principal axes, the programmer must generate the matrices needed to map an arbitrary axis to one of the principal axes, as discussed in Chapter 5.

The composition of the transformation elements is performed by SPHIGS at *traversal time*; thus, each time the display is regenerated, the composition must be performed. An alternative method for specifying a contiguous sequence of transformations increases the efficiency of the display-traversal process: Instead of making a structure element for each one, we compose them ourselves at *specification time* and generate a single transformation element. The following function does matrix multiplication at specification time:

```
matrix_4x4 SPH_composeMatrix (matrix_4x4 mat1, matrix_4x4 mat2, matrix_4x4 result);
```

The two setLocalTransformation elements in the preceding code can thus be replaced by

```
SPH_setLocalTransformation (
    SPH_composeMatrix (SPH_translate (...), SPH_rotateY (...), result), REPLACE);
```

Samsung/Google, EX1027, p. 321

The disadvantage of this method is that it is no longer possible to make a dynamic change to the size, orientation, or position of a primitive by selectively "editing" the desired member of the sequence of setLocalTransformation elements; rather, the entire composite must be recomputed and respecified. The rule of thumb for efficiency is thus to use composition at specification time unless the individual transformations are to be updated selectively, in which case they should be specified individually.

Let us create the street structure that contains three copies of our simple house, as first seen in Fig. 7.7. A perspective view of the "house" on the left, the "mansion" on the right, and the "cottage" in the middle is shown in Fig. 7.9(a). We have added dashed lines parallel to the x axis and tick marks for the x axis to indicate the relative positions of the houses, and have used a display mode of SPHIGS that shows the wireframe edges of polyhedra with hidden edges removed (see Section 7.8). The leftmost house in the figure is an untransformed instance of our standardized house, and the other two copies differ in size, orientation, and position.

The brute-force way to create this street structure is to specify the standardized house polyhedron three times, preceded by the desired transformation elements, as shown in the schematic representation of the structure of Fig. 7.9(b). We show a block of consecutive transformation elements as a single unit; the first element uses REPLACE mode and all others use PRECONCATENATION mode, with the multiplication symbol (·) separating them to indicate composition. The code generating the structure is shown in Fig. 7.10.

We can eliminate the redundant specifications of the standardized house polyhedron by defining a Pascal procedure to perform the call generating the house polyhedron, as shown in the pseudocode of Fig. 7.11. Because the house is defined by a single polyhedron call, the efficiency of this technique is not obvious in this example; however, if our house were

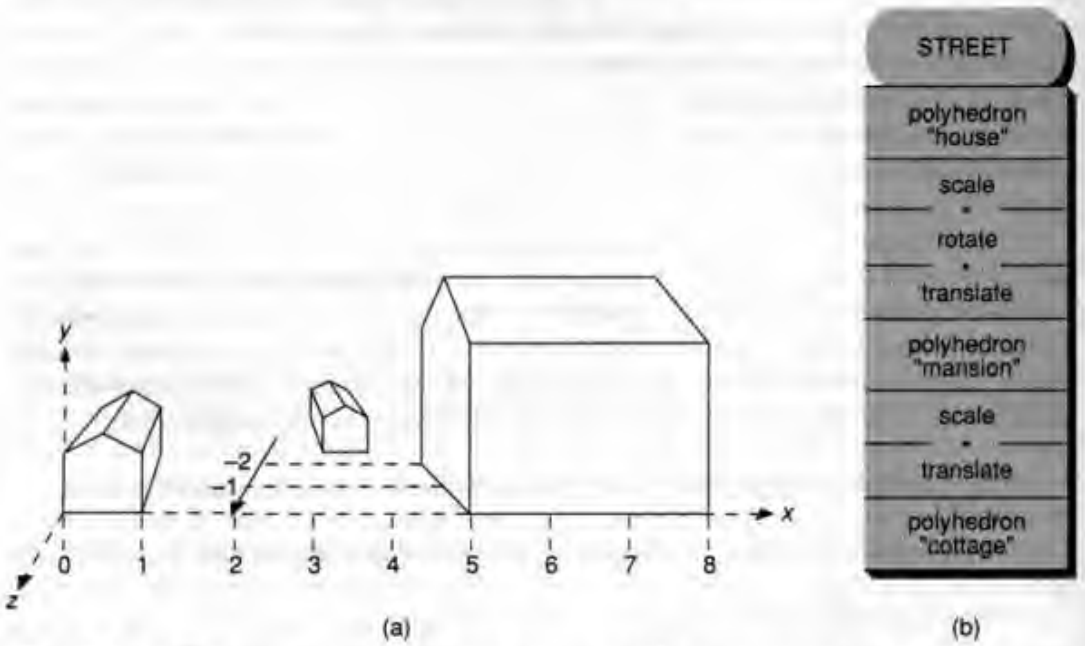


Fig. 7.9 Modeling a street with three houses. (a) Perspective view. (b) Structure.

Samsung/Google, EX1027, p. 322

```
SPH_openStructure (STREET_STRUCT);
    SPH_polyhedron (...);      /* Define first house, in its standardized form */

    /* Mansion is house scaled by 2 in x, 3 in y, 1 in z, rotated 90° about y, */
    /* then translated; note that its left side is subsequently front-facing */
    /* and lies in the (x, y) plane. */
    SPH_setLocalTransformation (SPH_scale (2.0, 3.0, 1.0, result), REPLACE);
    SPH_setLocalTransformation (SPH_rotateY (90.0, result), PRECONCATENATE);
    SPH_setLocalTransformation (SPH_translate (8.0, 0.0, 0.0, result), PRECONCATENATE);
    SPH_polyhedron (...);

    /* Cottage is house uniformly scaled by 0.75, unrotated, set back in z and over in x */
    SPH_setLocalTransformation (SPH_scale (0.75, 0.75, 0.75, result), REPLACE);
    SPH_setLocalTransformation (SPH_translate (3.5, 0.0, -2.5, result), PRECONCATENATE);
    SPH_polyhedron (...);
SPH_closeStructure ();
SPH_postRoot (STREET_STRUCT, 0);
```

Fig. 7.10 Code used to generate Fig. 7.9.

more complex and required a series of attribute and primitive specifications, this method clearly would require less C code. Moreover, the technique provides another benefit of modularization: We can change the house's shape or style by editing the House procedure without having to edit the code that creates the street.

We call a procedure such as House that generates a sequence of elements defining a standardized building block, and that is made to be used repetitively with arbitrary geometrical transformations, a *template procedure*. The template procedure is a convenience for the programmer and also represents good programming practice. Note, however, that although the House procedure adds a level of procedure hierarchy to the C program, no structure hierarchy is created—the model of the street is still "flat." Indeed, the structure network produced by the code of Fig. 7.11 is indistinguishable from that produced by the code of Fig. 7.10. There is no "savings" in terms of the number of elements produced for the structure.

```
void House (void)
{
    SPH_polyhedron (...);
}

/* Mainline */
SPH_openStructure (STREET_STRUCT);
    House ();                  /* First House */
    set local transformation matrix;
    House ();                  /* Mansion */
    set local transformation matrix;
    House ();                  /* Cottage */
SPH_closeStructure ();
SPH_postRoot (STREET_STRUCT, 0);
```

Fig. 7.11 Use of a template procedure to model the street.

Samsung/Google, EX1027, p. 323

One change we could make to our template procedure is to have it accept a transformation matrix as a parameter, which it would then use to specify a setLocalTransformation element.[7] Although in some cases passing transformation parameters would be convenient, this method lacks the generality inherent in our original method of being able to specify an arbitrary number of transformations before calling the template.

7.5 HIERARCHICAL STRUCTURE NETWORKS

7.5.1 Two-Level Hierarchy

So far, we have dealt with three types of structure elements: output primitives, appearance attributes, and transformations. Next, we show how the power of SPHIGS derives in large part from structure hierarchy, implemented via an element that "calls" a substructure when executed during traversal. Structure hierarchy should not be confused with the template-procedure hierarchy of the previous section. Template-procedure hierarchy is resolved at specification time, when the CSS is being edited, and produces in-line elements, not substructure invocations. By contrast, structure hierarchy induced by invocation of substructures is resolved when the CSS is traversed for display—the execution of an invocation element acts as a subroutine call. In Section 7.15, we take another look at the relative advantages and disadvantages of template-procedure hierarchy and structure hierarchy.

The *structure-execution element* that invokes a substructure is created by

void SPH_executeStructure (**int** *structureID*);

Let us replace the template procedure of the previous section by a procedure that builds a house structure in the CSS (see Fig. 7.12). This procedure is called exactly once by the mainline procedure, and the HOUSE_STRUCT is never posted; rather, its display results from its being invoked as a subobject of the street structure. Note that the only differences in the STREET_STRUCT specification are the addition of the call to the procedure that builds the house structure and the replacement of each template procedure call by the generation of an execute-structure element. Although the displayed image is the same as that of Fig. 7.9(a), the structure network is different, as shown in Fig. 7.13, in which the execute-structure element is depicted as an arrow.

Posting STREET_STRUCT tells SPHIGS to update the screen by traversing the STREET_STRUCT structure network; the traversal is in the form of a depth-first tree walk, just as a procedure/subroutine hierarchy is executed. In the preceding example, the traverser initializes the street structure's local matrix to the identity matrix, and then performs the first invocation of the house substructure, applying the street structure's local matrix to each of the house's vertices as though the house polyhedron were a primitive element in the street structure itself. When it returns from the first invocation, it sets the local matrix to a desired composite transformation, and then performs the second invocation, applying the

[7] Newman defines *display procedures* as template procedures that take scaling, rotation, and translation parameters [NEWM71].

Samsung/Google, EX1027, p. 324

```
void BuildStandardizedHouse (void)
{
    SPH_openStructure (HOUSE_STRUCT);
        SPH_polyhedron (...);
    SPH_closeStructure ();
}

/* Mainline */
BuildStandardizedHouse ();
SPH_openStructure (STREET_STRUCT);
    SPH_executeStructure (HOUSE_STRUCT);     /* First house */
    set local transformation matrix;
    SPH_executeStructure (HOUSE_STRUCT);     /* Mansion */
    set local transformation matrix;
    SPH_executeStructure (HOUSE_STRUCT);     /* Cottage */
SPH_closeStructure ();
SPH_postRoot (STREET_STRUCT, 0);
```

Fig. 7.12 Use of a subordinate structure to model the street.

new composite matrix to the vertices of the house to create the second instantiation of the house. When it returns, the local matrix is again changed; the new composite is applied to the house's vertices to produce the third house instance.

We think of a structure as an independent object, with its primitives defined in its own floating-point modeling-coordinate system (MCS); this way of thinking facilitates the building of low-level standardized building blocks. As we noted in Section 5.8, a transformation maps the vertices in one coordinate system into another; here, SPHIGS uses the local matrix of structure S to transform the primitives of substructures into S's own MCS.

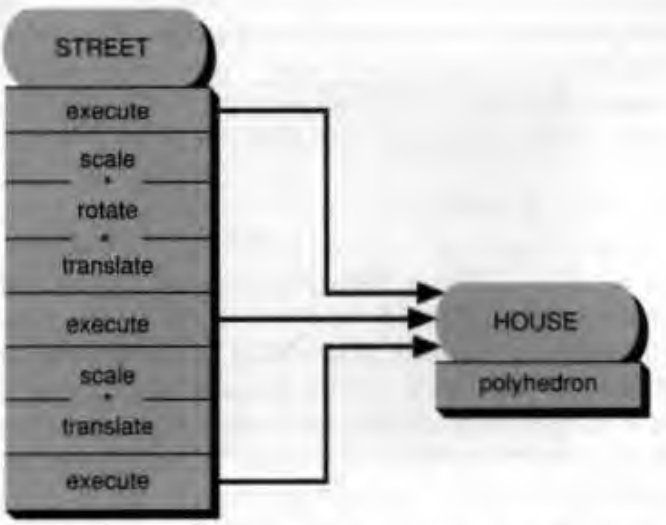


Fig. 7.13 Structure network showing invocation of subordinate structure.

Samsung/Google, EX1027, p. 325

7.5.2 Simple Three-Level Hierarchy

As a simple example of three-level hierarchy, we extend the house in our street example. The new house is composed of the original standardized house (renamed SIMPLE_HOUSE_STRUCT) and a chimney suitably scaled and translated to lie in the right place on top of the house. We could revise the house structure by adding the chimney's facets directly to the original polyhedron, or by adding a second polyhedron to the structure, but we choose here to induce three-level hierarchy by decomposing the house into two subobjects. An advantage of this modularization is that we can define the chimney in a standardized manner (at the origin, of unit size) in its own MCS (as shown in Fig. 7.14a),

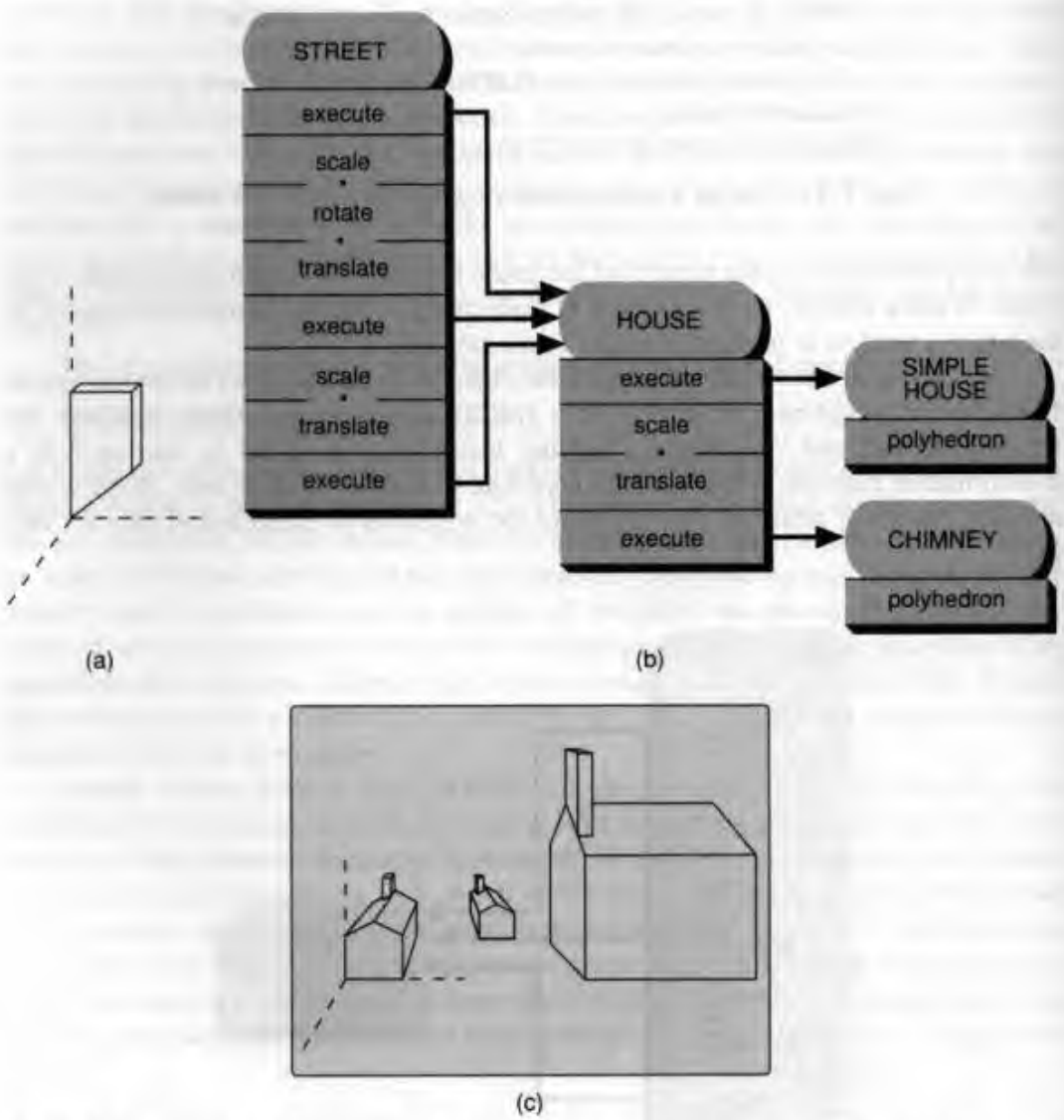


Fig. 7.14 Three-level hierarchy for street. (a) Standardized chimney. (b) Structure network. (c) Resulting image.

Samsung/Google, EX1027, p. 326

and then use scaling and translation to place it on the roof in the house's MCS. If we had to define the chimney to fit exactly on the roof and to map into the house's MCS without scaling, we would have to do messy calculations to compute the vertices explicitly. With modularity, however, we simply define the standardized chimney such that its bottom facet has the same slope as the roof itself; with that condition met, uniform scaling and arbitrary translation can be applied.

The revised house structure is built via

```
SPH_openStructure (HOUSE_STRUCT);
    SPH_executeStructure (SIMPLE_HOUSE_STRUCT);
    set local matrix to scale/translate standardized chimney onto roof of simple house;
    SPH_executeStructure (CHIMNEY_STRUCT);
SPH_closeStructure ();
```

What happens when this two-level house object is instantiated by the street structure with a transformation to yield the three-level hierarchy shown in Fig. 7.14(b)? Since SPHIGS transforms a parent by transforming the latter's component elements and substructures, we are assured that the two component primitives (the simple house and the chimney) are transformed together as a single unified object (Fig. 7.14c). The key point is that the street-structure specification did not need to be changed at all. Thus, the designer of the street structure does not need to be concerned with the internal details of how the house is constructed or subsequently edited—it is a black box.

7.5.3 Bottom-Up Construction of the Robot

Let us now look at a more interesting example, our simple robot, which combines the key ideas of modeling using structure hierarchy and of repeated editing of transformations to achieve motion dynamics. A complex object or system hierarchy is usually conceptualized and informally described top-down. For example, a computer-science department building is composed of floors, which are composed of offices and laboratories, which are composed of furniture and equipment, and so on. Recall that our simple android robot is composed of an upper body and a pedestal base; the upper body is composed of a trunk, a head, and two identical arms, each of which is composed of a fixed hand and a "sliding" (translating) thumb as gripper.

Even though we design top-down, we often implement bottom-up, defining building blocks for use in the definitions of higher-level building blocks, and so on, to create the hierarchy of building blocks. Thus, in constructing the robot, we define the thumb building block for the robot's arm, then the arm itself, and then join two instances of the arm building block onto the upper body, and so on, as shown in the symbolic parts hierarchy of Fig. 7.2 and in the more detailed structure network diagram of the upper body in Fig. 7.15.

Let us look at the bottom-up construction process in more detail to see what geometry and transformations are involved. It makes sense to design the arm and thumb in the same units of measurement, so that they fit together easily. We define the thumb structure in a standardized position in its own MCS in which it "hangs" along the y axis (Fig.7.16a). The arm structure is defined with the same unit of measurement as that used for the thumb;

Samsung/Google, EX1027, p. 327

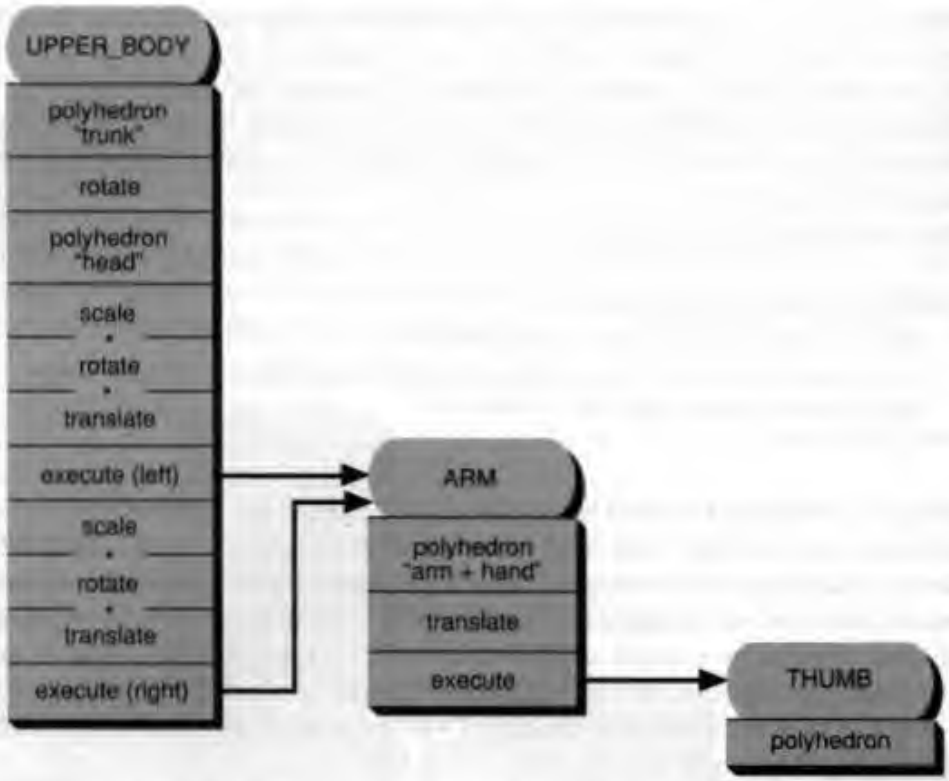


Fig. 7.15 Structure hierarchy for robot's upper body.

it consists of the arm+hand polyhedron (standardized, hanging down along the *y* axis, as shown in Fig. 7.16b) and a translated invocation of the thumb structure. The translation element preceding the invocation of the thumb is responsible for moving the thumb from its standardized position at the origin to its proper place on the wrist of the arm (c).

The arm-invoking-thumb network is a two-level hierarchy similar to the street–house example. By editing the translation element in the arm structure, we can "slide" the thumb along the wrist of the arm (Fig. 7.16d).[8]

Next, we build the upper body. Since we want to be able to rotate the head, we first specify the trunk polyhedron, then a rotation, and next the head polyhedron (Fig. 7.16e). Our next step is to have the upper-body structure invoke the arm structure twice. What transformations should precede these invocations? If our sole concern is positioning (i.e., translating) each arm correctly in the upper-body MCS, we may produce a picture like Fig. 7.16(f), where the arm and upper body were clearly designed at different scales. This is easy to fix: We can add a scale transformation preceding the translation (Fig. 7.16g). However, a scale and a translation is not enough if we want arms that can swing on the axis

[8] The observant reader may have wondered how the thumb actually slides in this model, since it is not really attached in any way to the arm hand. In fact, none of the components of our robot model is attached to another via objects representing joints. Modeling of joints and the constraints that determine their modes of operation are beyond the scope of this book.

Samsung/Google, EX1027, p. 328

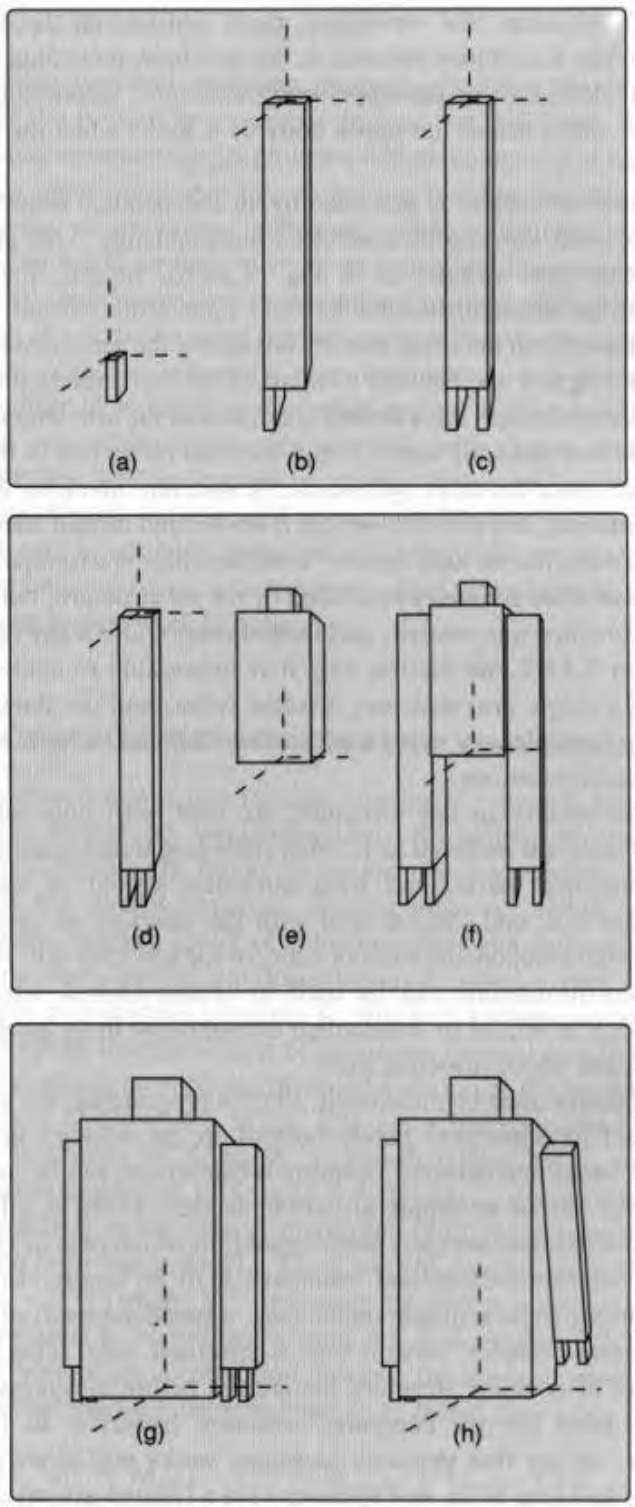


Fig. 7.16 Constructing the robot's upper body. (a) Thumb. (b) Fixed arm plus hand. (c) Completed arm. (d) Completed arm with translated thumb. (e) Trunk and head. (f) Upper body with outsized arms. (g) Corrected arms. (h) Left arm rotated.

Samsung/Google, EX1027, p. 329

connecting the two shoulders (for movement much like that of the arm of a marching soldier); for this, we place a rotation element in the structure, preceding the translation. We have completed our definition of the upper-body structure; assembling the full robot is Exercise 7.1. Fig. 7.16(h) shows the upper body as it looks when the left arm's rotation element is nonzero.

Because each arm invocation is preceded by an independent sequence of transformations, the motion of each arm can be controlled independently. One arm can be hanging down while the other arm swings, as in Fig. 7.16(h). Indeed, the difference in the transformations is what distinguishes the left and right arms. Notice, however, that the movable thumb is not only on the same side of, but also is the same distance from, the fixed hand on both left and right arms, because it is part of the arm's internal definition. (In fact, if the application simply changes the translation element in the arm structure, all the thumbs on all the robot instances suddenly move!) Thus we must rotate one of the arms 180° about the y axis in order to make the arms symmetric. A structure invoking the arm can control the arm's size, orientation, and position only *as a whole*, and cannot alter the arm's internal construction in any way. As we said earlier, a substructure is essentially a black box; the invoker needs to know what geometry is defined by the substructure, but it does not need to know how the substructure was created, and indeed cannot affect any of the substructure's internals. In Section 7.14.2, we discuss why it is impossible to control the two thumbs independently with a single arm structure invoked twice, and we show how to solve this problem by creating multiple arm structures that are identical in form but have potentially differing internal transformations.

In summary, in specifying any structure, we deal with only what primitives and lower-level substructures are included in it, what modeling transformations should be used to position its component parts, and what attributes should be used to affect their appearance. We need not, and cannot, deal with the internals of lower-level structures. Furthermore, we design components without concern for how they will be used by invoking structures, since transformations can be used to obtain desired size, orientation, and position. In practice, it is helpful to standardize a component in its local MCS, so that it is easy to scale and rotate about principal axes.

Two additional points must be mentioned. First, a programmer does not need to design purely top-down and to implement purely bottom-up; by analogy to the programming technique of using "stubs and drivers," dummy substructures can be used in the structure hierarchy. A dummy can be an empty structure; in fact, SPHIGS allows a structure to execute a substructure that has not (yet) been created, in which case SPHIGS automatically creates the referenced substructure and initializes it to be empty. In some cases, it is desirable for the dummy to be a simple object (say, a parallelepiped) of approximately the same size as the more complex version that is specified later. This technique permits top-down debugging of complex structure hierarchies before all components are defined fully. Second, we have not yet compared structure hierarchy to template-procedure hierarchy. For now, we say that structure hierarchy works well if we need to instantiate multiple copies of a building block and to control (to a limited extent) the appearance and placement of each copy but not its internal definition. We discuss the tradeoffs between template procedure and structure hierarchy more fully in Section 7.15.

Samsung/Google, EX1027, p. 330

7.5.4 Interactive Modeling Programs

Interactive 3D construction and modeling programs facilitate the construction of object hierarchies through the bottom-up assembly process just described. Most such programs offer a palette of icons representing the program's fundamental set of building blocks. If the drawing program is application-specific, so are the building blocks; otherwise, they are such common atoms as polylines, polygons, cubes, parallelepipeds, cylinders, and spheres. The user can select an icon in order to instantiate the associated building block, and can then specify transformations to be applied to the building-block instance. Such specification is typically done via input devices, such as the mouse or control dials, that let the user experiment with the instance's size, orientation, and position until it "looks right." Since it is difficult to judge spatial relationships in 2D projections of 3D scenes, more sophisticated interaction techniques use 3D grids, often with "gravity fields" surrounding intersection points, numerical scales, sliders of various types, numerical feedback on the position of vertices, and so on (see Section 8.2.6). Some construction programs allow the user to combine instances of fundamental graphical building blocks to create a higher-level building block, which is then added to the building-block palette, to be used in building even higher-level objects.

7.6 MATRIX COMPOSITION IN DISPLAY TRAVERSAL

So far, we have discussed how a programmer constructs a model, using top-down design and (more or less) bottom-up implementation. Regardless of how the model was constructed, SPHIGS displays the model by performing a top-down, depth-first search of the DAG rooted at the posted structure. During traversal, SPHIGS processes all the geometry specified by multiple levels of transformation and invocation. To see what is involved, we observe that, during top-down traversal, when a root structure A invokes structure B, which in turn invokes structure C, this is is tantamount to saying that B was constructed bottom-up by transformation of primitives defined in C's MCS to B's MCS, and that A was then similarly constructed by transformation of B's primitives (including any defined via invocation of C) to A's MCS. The net effect was that C's primitives were transformed twice, first from MCS_C to MCS_B and then from MCS_B to MCS_A.

Using the notation developed in Section 5.8, let $M_{B \leftarrow C}$ denote the value of the local matrix for structure B that, at the time of invocation of C, maps vertices in MCS_C to their properly transformed positions in MCS_B. Thus, to map a vertex from MCS_C to MCS_B, we write $V^{(B)} = M_{B \leftarrow C} \cdot V^{(C)}$ (where $V^{(H)}$ indicates the vector representing a vertex whose coordinates are expressed in coordinate-system H), and similarly, $V^{(A)} = M_{A \leftarrow B} \cdot V^{(B)}$. Thus, to mimic the bottom-up construction process, the traverser must successively apply the transformations that map the vertices from C to B and then from B to A:

$$V^{(A)} = M_{A \leftarrow B} \cdot V^{(B)} = M_{A \leftarrow B} \cdot (M_{B \leftarrow C} \cdot V^{(C)}) \tag{7.1}$$

By matrix associativity, $V^{(A)} = (M_{A \leftarrow B} \cdot M_{B \leftarrow C}) \cdot V^{(C)}$. Therefore, the traverser simply composes the two local matrices and applies the resulting matrix to each of C's vertices.

Samsung/Google, EX1027, p. 331

Using tree notation, let the root be at level 1 and the successive children be at levels 2, 3, 4, ... Then, by induction, for any structure at level j ($j > 4$), we can transform a vertex $V^{(j)}$ in that structure's MCS into the vertex $V^{(1)}$ in the root coordinate system via

$$V^{(1)} = (M_{1\leftarrow 2} \cdot M_{2\leftarrow 3} \cdot \ldots \cdot M_{(j-1)\leftarrow j}) \cdot V^{(j)} \tag{7.2}$$

Since SPHIGS allows primitives to be transformed within the local MCS with the local matrix, a vertex $V^{(j)}$ is obtained by applying the local matrix to the coordinate values of the primitive:

$$V^{(j)} = M^{(j)} \cdot V^{(\text{prim})} \tag{7.3}$$

We use $M^{(j)}$ to denote the local matrix while the structure is being traversed to show that the matrix is being used to transform primitives into the structure's own level-j MCS. If the structure subsequently invokes a subordinate, the matrix's use changes; it is then used to transform the invoked structure at level $j + 1$ into the level-j MCS, and we denote it with $M_{j\leftarrow(j+1)}$. This does not imply that the matrix's value changes—only its use does.

Combining Eqs. (7.2) and (7.3) using associativity, we get

$$V^{(1)} = (M_{1\leftarrow 2} \cdot M_{2\leftarrow 3} \cdot \ldots \cdot M_{(j-1)\leftarrow j} \cdot M^{(j)}) \cdot V^{(\text{prim})} \tag{7.4}$$

Thus, to transform a primitive at level j in the hierarchy to the MCS of the root (which is the world-coordinate space), all we need to do is to apply the composition of the current values of the local matrix for each structure from the root down to the structure in which the primitive is defined. This composite of local matrices—the term in parentheses in Eq. (7.4)—is called the *composite modeling transformation matrix* (CMTM). When the state of a traversal is such that a level-j structure's elements are being executed, the CMTM is the composition of j matrices. Only the last of those matrices ($M^{(j)}$) may be changed by the structure, because a structure may modify only its local matrix.[8] Thus, while the structure is active, the first $j - 1$ matrices in the CMTM list are constant. The composite of these $j - 1$ matrices is the *global matrix* (GM)—the term in parentheses in Eq. (7.2)—for the level-j structure being executed. It is convenient for SPHIGS to maintain the GM during the traversal of a structure; when a setLocalTransformation element modifies the local matrix (LM), SPHIGS can easily compute the new CMTM by postconcatenating the new local matrix to the GM.

We can now summarize the traversal algorithm, to be elaborated in Section 7.12.1. SPHIGS does a depth-first traversal, saving the CMTM, GM, and LM just before any structure invocation; it then initializes the substructure's GM and CMTM to the inherited CMTM, and its LM to the identity matrix. The CMTM is applied to vertices and is updated by changes to the LM. Finally, when the traverser returns, it restores the CMTM, GM, and LM of the parent structure and continues. Because of the saving and restoring of the matrices, parents affect their children but not vice versa.

---

[8] We present the global matrix as a derived entity that cannot be modified by a structure. In true PHIGS, a structure can modify the GM active during its execution, but this power is still "localized" in the sense that it in no way affects the local matrices of its ancestors.

Samsung/Google, EX1027, p. 332

Let us watch as SPHIGS traverses the three-level upper-body–arm–thumb hierarchy of Fig. 7.15. We have posted the UPPER_BODY structure as a root. Figure 7.17(a) shows a sequence of snapshots of the traversal state; a snapshot has been created for each point marked with a number in the structure network diagram of Fig. 7.17(b).

The traversal state is maintained via a stack, shown in Fig. 7.17(a) growing downward, with the currently active structure in a solid rectangle and its ancestors in dashed ones. The values of the three state matrices for the currently active structure are shown to the right of the stack diagram. Arcs show the scope of a transformation: The GM arc illustrates that a structure's ancestors contribute to the GM, and the CMTM arc shows that the CMTM is the product of the GM and LM. Recall that in each group of transformations, the first is in REPLACE mode and the rest are PRECONCATENATEd. Thus the first "rotate" in the structure applies only to the head since it will be REPLACEd by the first "scale" that applies to the left arm.

At point 1 in Fig. 7.17(b), the traverser is about to execute the first element in the root. Because a root has no ancestors, its GM is identity; the LM is also identity, as is the case whenever a structure's execution begins. At point 2, the LM is set to the composite of the (Scale, Rotate, Translate) transformation triplet. Therefore, the CMTM is updated to the product of the identity GM and the SRT composite, and is then ready for use in transforming the arm subobject to the upper body MCS to become the left arm via $(SRT)_{ub \leftarrow la}$. Next, at point 3, the traverser is about to execute the first element of the arm structure. The GM for the arm execution is, as one would expect for a level-2 instantiation, its parent's LM at the point of invocation.

At point 4, the arm LM is set to position the thumb within the arm ($T_{arm \leftarrow th}$), and the CMTM is updated to the product of the GM and LM. This level-2 CMTM becomes the GM for the level-3 instantiation of the thumb (point 5). Since the LM of the thumb is identity, the CMTM of the thumb has the desired effect of transforming thumb coordinates first to arm coordinates, then to upper-body coordinates. At point 6, the traverser has returned from the thumb and arm invocations, back to the upper body. The matrices for the upper-body structure are as they were before the invocation, since its subordinates cannot change its local matrix. At point 7, the LM of the upper body is replaced with a new composite for the right arm. When we have descended into the thumb structure for the right arm (point 8), the CMTM is almost identical to that at point 5; the only difference is the level-2 matrix that moves the arm into position on the upper body.

To animate a composite object such as the robot, we need only to think about how each child structure is to be affected in its parent, and to define the appropriate transformation elements for each component that can be edited dynamically later. Thus, to make the robot spin about its axis, raise its arm, and open its hand, we change the rotation matrix in the robot structure to affect the upper body, the rotation matrix in the upper-body structure to affect the arm, and the translation matrix in the arm structure to affect the thumb. The transformations are done independently at each level of the hierarchy, but the net effect is cumulative. The difficult part of specifying an animation is working backward from a desired result, such as "the robot moves to the northwest corner of the room and picks up a block from the table," to derive the sequence of transformations yielding that result. Chapter 21 briefly discusses the issues involved in such "inverse kinematics" problems.

Samsung/Google, EX1027, p. 333

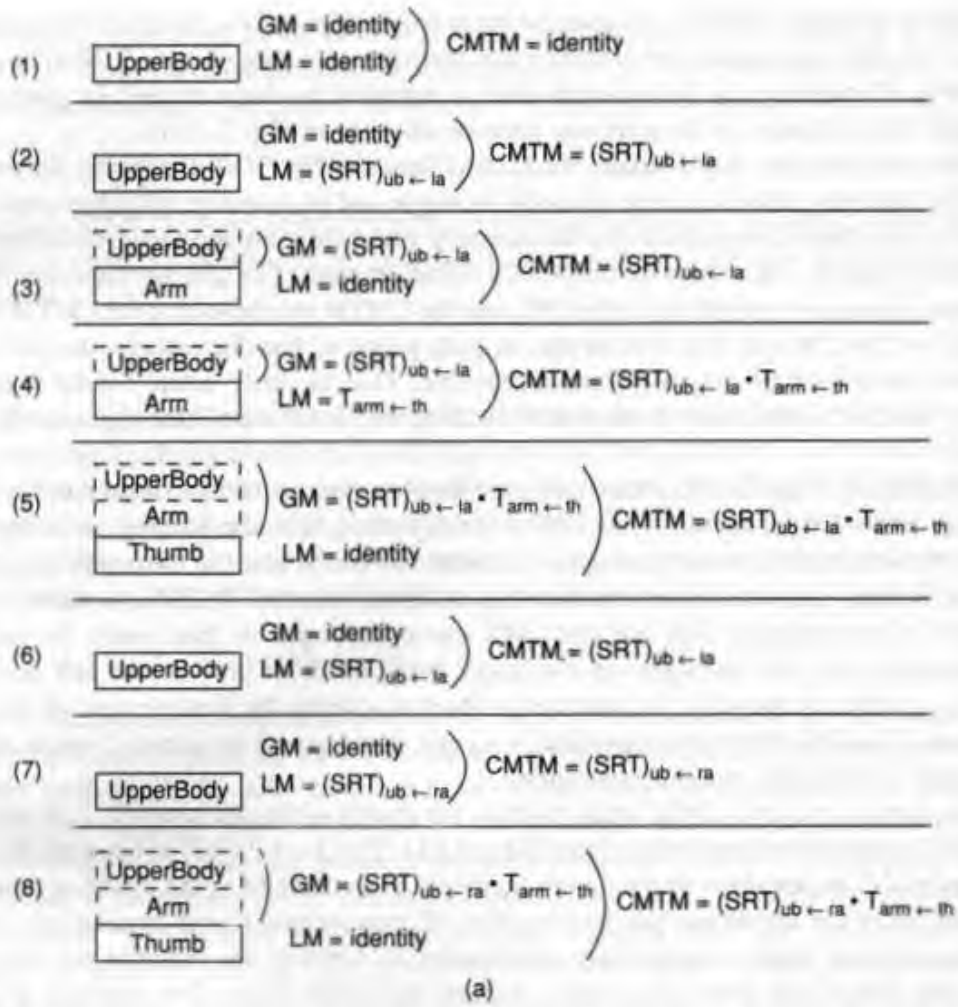


Fig. 7.17 Traversal of a three-level hierarchy. (a) Snapshots of traversal-state stack. (b) Annotated structure network.

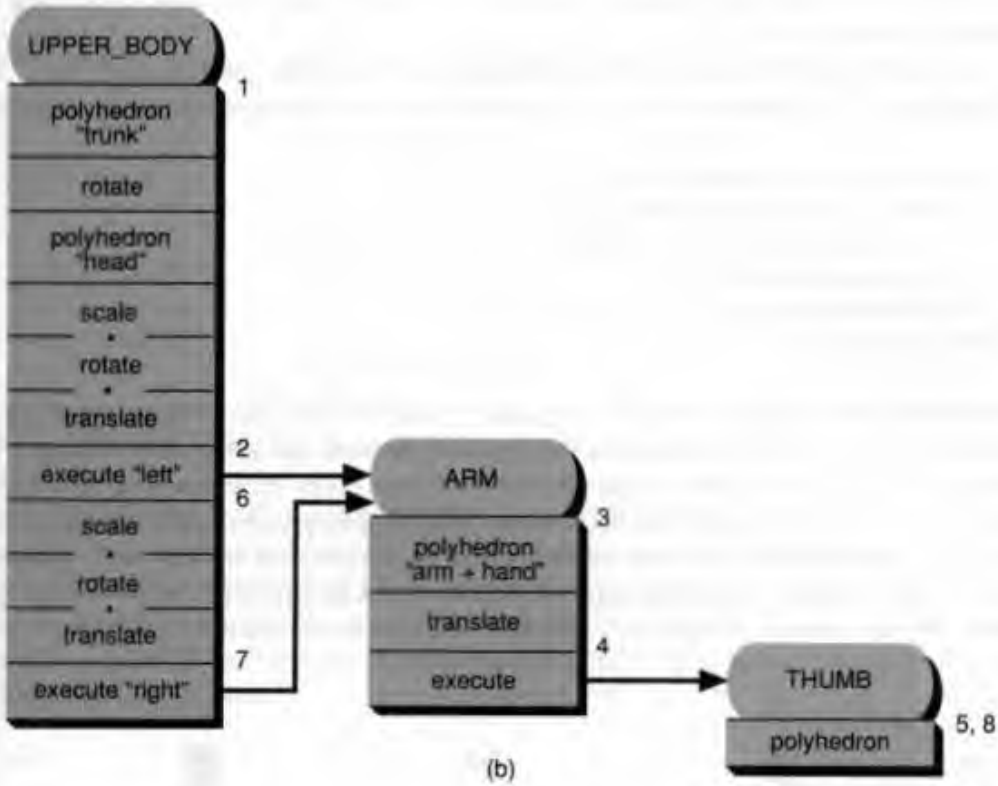
7.7 APPEARANCE-ATTRIBUTE HANDLING IN HIERARCHY

7.7.1 Inheritance Rules

The attribute-traversal state is set by attribute elements during traversal, and, as in SRGP, is applied modally to all primitives encountered. We saw how parents affect their children via geometric transformations. What rules pertain to appearance attributes? In our street example, the houses all have the default color. To give an object a particular color (e.g., to make a house brown), we can specify that color as an initial element in the object structure itself, but that makes the object's color intrinsic and not changeable during traversal. We

Samsung/Google, EX1027, p. 334

Fig. 7.17 *(Cont'd.)*



would prefer to "pass the color as a parameter," so that the child can inherit it the way a child inherits the CMTM as its GM.

Indeed, in SPHIGS, each substructure inherits the traversal state as the latter exists at the time of the invocation of the substructure, and can then modify that state at will without affecting its ancestors. In other words, attributes and transformations are bound dynamically at traversal time, rather than statically at specification time. This dynamic binding is one of the major features of SPHIGS, making it easy to customize instances of a substructure.

What substructures do with the inherited state depends on the type of data involved. We saw that, for geometric transformations, the substructure inherits the GM but cannot override its inheritance, since it can affect only its own local matrix. Attributes are simpler in that the substructure inherits the parent's attributes as the initial values of its local attribute state, but can change its local state subsequently. There is no need to distinguish between global and local attributes, since there is no notion of composing attributes. Note that this mechanism has the same problem we discovered with transformation inheritance—just as our robot's two arm instances cannot have differing thumb transformations, its two arm instances cannot have the same color for the fixed part but differing colors for the thumb.

Samsung/Google, EX1027, p. 335

In the structure network of Fig. 7.18(a), the street structure sets the colors for the house substructure. The resulting image is shown in Fig. 7.18(b), and the code generating the network is shown in Fig. 7.19.

An attribute can be reset within a substructure to override the inherited value. The following code fragment specifies a revised house structure whose chimney is always red.

```
SPH_openStructure (HOUSE_STRUCT);
    SPH_executeStructure (SIMPLE_HOUSE_STRUCT);
    SPH_setInteriorColor (COLOR_RED);
    set up transformation;
    SPH_executeStructure (CHIMNEY_STRUCT);
SPH_closeStructure ();
```

Let us use this new house structure in conjunction with the street structure generated by the code in Fig. 7.19. Figure 7.20 shows the structure network and the resulting image. The traverser starts at STREET_STRUCT; the interior- and edge-color attributes have their default values. The edge color is set to white, a value it retains throughout display traversal of this network. The first setInteriorColor causes yellow to be inherited by the first instance of HOUSE_STRUCT, which in turn passes yellow to SIMPLE_HOUSE_STRUCT, whose polyhedron is shown in that color. When the traverser returns from SIMPLE_HOUSE_STRUCT to

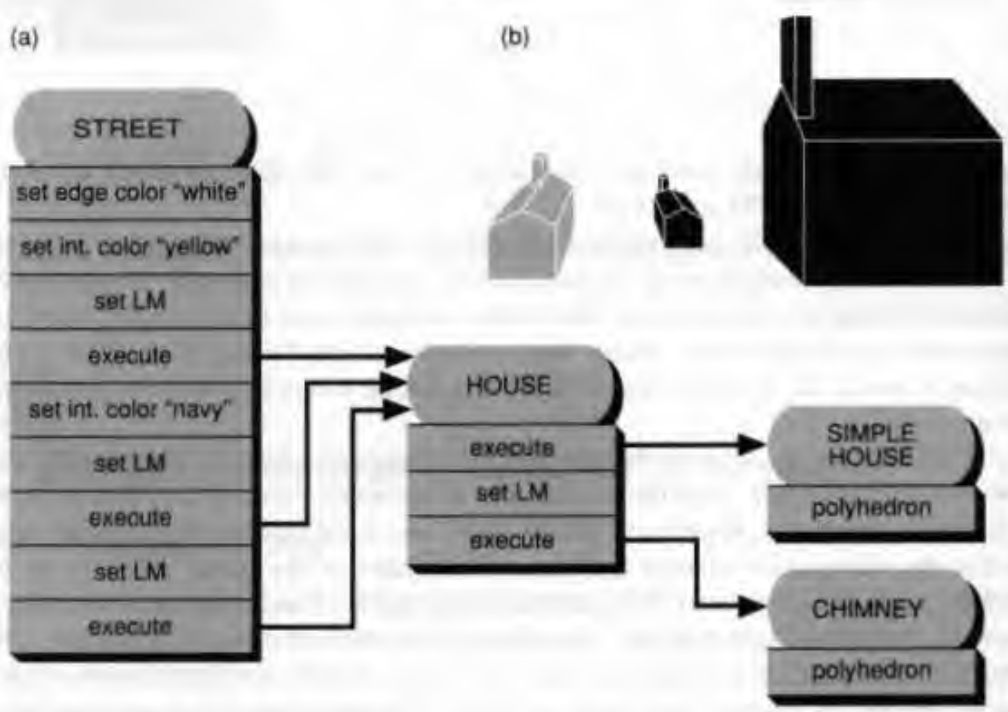


Fig. 7.18 Use of attribute inheritance to model street with colored houses. (a) Structure network. (b) Resulting image. (Interior colors are simulated by patterns.)

Samsung/Google, EX1027, p. 336

```
SPH_openStructure (STREET_STRUCT);
    SPH_setEdgeColor (COLOR_WHITE);

    SPH_setInteriorColor (COLOR_YELLOW);
    set up transformation;
    SPH_executeStructure (HOUSE_STRUCT);

    SPH_setInteriorColor (COLOR_NAVY);
    set up transformation;
    SPH_executeStructure (HOUSE_STRUCT);

    set up transformation;
    SPH_executeStructure (HOUSE_STRUCT);
SPH_closeStructure ();
```

Fig. 7.19 Code used to generate Fig. 7.18.

HOUSE_STRUCT, the interior-color attribute is immediately changed to red by the next element. The invocation of CHIMNEY_STRUCT therefore results in a red chimney with white edges. None of these operations affect the attribute group for STREET_STRUCT, of course; when the traverser returns from HOUSE_STRUCT, STREET_STRUCT's interior-color attribute is restored to yellow. The interior-color attribute is then changed to navy to prepare for drawing two navy houses.

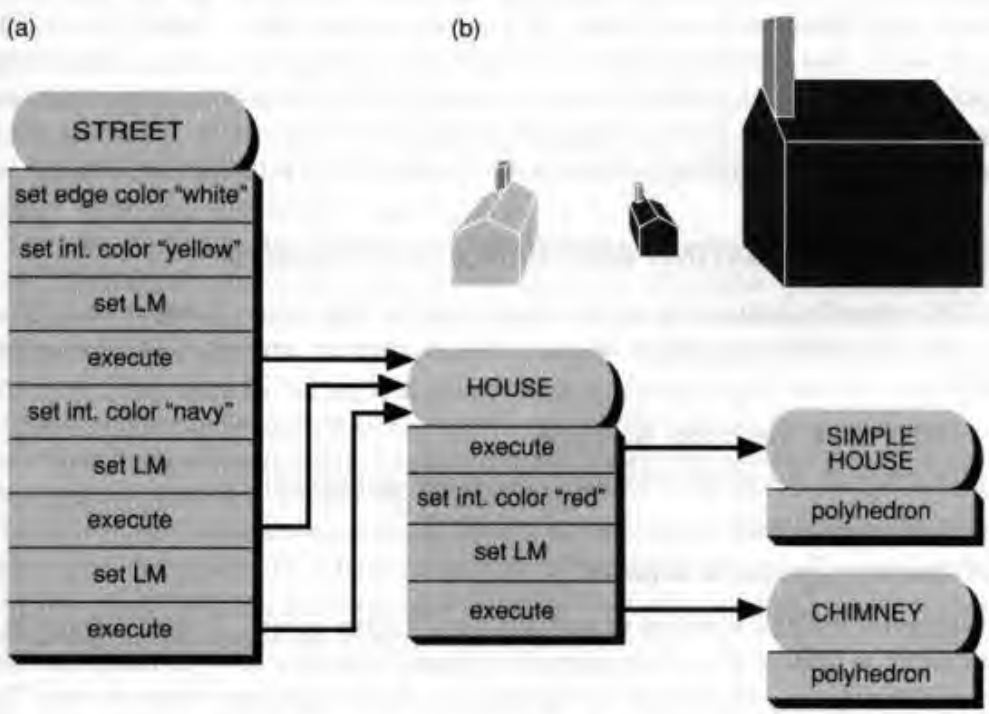


Fig. 7.20 Subordinate structure overriding an inherited attribute. (a) Structure network. (b) Resulting view.

Samsung/Google, EX1027, p. 337

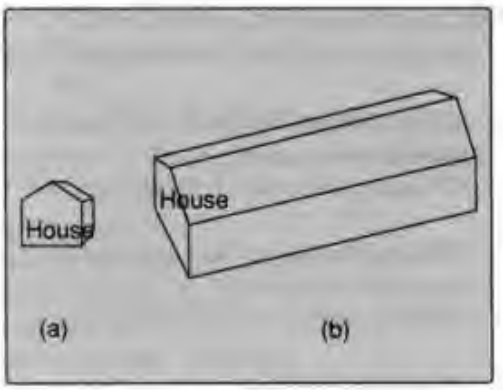


Fig. 7.21 The nongeometric nature of text in SPHIGS. (a) Before transformation. (b) After transformation.

7.7.2 SPHIGS Attributes and Text Unaffected By Transformations

In true PHIGS implementations, text can be subjected to transformations like any other primitive. Thus, the text characters on the sides of a truck, displayed in perspective, are rotated and shown with appropriate perspective foreshortening, as though the letters were made out of individual polylines or fill areas. Similarly, dashes in dashed lines should be subject to geometric transformations and perspective foreshortening. However, just as attributes in SPHIGS are nongeometric for performance reasons, so is text. As in SRGP, the font of the text determines the text's screen size, and a text string cannot even be rotated—the image of a text string is always upright in the plane of the screen, and is never compressed or expanded. Thus, rotation and scaling affect text's origin, but not its size and orientation (Fig. 7.21). SPHIGS primitive text is thus useful primarily for labeling.

7.8 SCREEN UPDATING AND RENDERING MODES

SPHIGS constantly updates the screen image to match the current status of the CSS and view table. The following actions all can make an arbitrary amount of the screen image obsolete:

- An entry in the view table is changed
- A structure is closed (after having been opened and edited)
- A structure is deleted
- A structure is posted or unposted.

Whenever SPHIGS is called to perform one of these actions, it must regenerate the screen image to display the current state of all posted networks. How SPHIGS chooses to generate the image is a function of the rendering mode the application has selected. These modes present a choice between quality and speed of regeneration: The higher the quality, the longer it takes to render the image. The rendering mode is set by

procedure SPH_setRenderingMode (*mode* : WIREFRAME / FLAT / LIT_FLAT / GOURAUD);

Samsung/Google, EX1027, p. 338

We summarize the four SPHIGS rendering modes here; they are discussed much more fully in Chapters 14 through 16.

Wireframe rendering mode. WIREFRAME mode is the fastest but least realistic form of display. Objects are drawn as though made of wire, with only their edges showing. The visible (within the view volume) portions of all edges of all objects are shown in their entirety, with no hidden-edge removal. Primitives are drawn in temporal order—that is, in the order in which the traverser encounters them in the posted structure networks in the database; this order is affected by the display-priority determined by the view index, as mentioned in Section 7.3.4.

All edge attributes affect screen appearance in their designated way in this mode; in fact, when the edge flag is set to EDGE_INVISIBLE, fill areas and polyhedra are entirely invisible in this mode.

Shaded rendering modes. In its other three rendering modes, SPHIGS displays fill areas and polyhedra in a more realistic fashion by drawing fill areas and facets as filled polygons. The addition of shaded areas to the rendering process increases the complexity significantly, because spatial ordering becomes important—portions of objects that are hidden (because they are obscured by portions of "closer" objects) must not be displayed. Methods for determining visible surfaces (also known as hidden-surface removal) are discussed in Chapter 15.

For the three shaded rendering modes, SPHIGS "shades" the interior pixels of visible portions of the facets; the quality of the rendering varies with the mode. For FLAT shading, the mode used often in this chapter's figures, all facets of a polyhedron are rendered in the current interior color, without being influenced by any light sources in the scene. Visible portions of edges are shown (if the edge flag is EDGE_VISIBLE) as they would appear in WIREFRAME mode. If the interior color is set to match the screen background, only the edges show—this use of FLAT rendering, which produced Figs. 7.9(a) and 7.14(c), simulates wireframe with hidden-edge removal.

The two highest-quality rendering modes produce images illuminated by a light source;[10] illumination and shading models are discussed in Chapter 16. These images are nonuniformly "shaded;" the colors of the pixels are based on, but are not exactly, the value of the interior-color attribute. In LIT_FLAT mode, all the pixels on a particular facet have the same color, determined by the angle at which the light hits the facet. Because each facet is of a uniform color, the image has a "faceted" look, and the contrast between adjacent faces at their shared edge is noticeable. GOURAUD mode colors the pixels to provide a smooth shaded appearance that eliminates the faceted look.

In FLAT mode, the edge-flag attribute should be set to EDGE_VISIBLE, because, without visible edges, the viewer can determine only the silhouette boundary of the object. In the two highest-quality modes, however, edge visibility is usually turned off, since the shading helps the user to determine the shape of the object.

---

[10] The PHIGS+ extension provides many facilities for controlling rendering, including specification of the placement and colors of multiple light sources, of the material properties of objects characterizing their interaction with light, and so on; see Chapters 14 through 16.

Samsung/Google, EX1027, p. 339

7.9 STRUCTURE NETWORK EDITING FOR DYNAMIC EFFECTS

As with any database, we must be able not only to create and query (in order to display) the SPHIGS structure database, but also to edit it in a convenient way. An application edits a structure via the procedures described in this section; if the application also maintains an application model, it must ensure that the two representations are edited in tandem. *Motion dynamics* requires modification of viewing or modeling transformations; *update dynamics* requires changes in or replacement of structures. The programmer may choose to edit a structure's internal element list if the changes are relatively minor; otherwise, for major editing, it is common to delete and then to respecify the structure in its entirety.

In the remainder of this section, we present methods for intrastructure editing; see the SPHIGS reference manual for information on editing operations that affect entire structures (e.g., deletion, emptying), and for more detailed descriptions of the procedures presented here.

7.9.1 Accessing Elements with Indices and Labels

The rudimentary editing facilities of both SPHIGS and PHIGS resemble those of old-fashioned line-oriented program editors that use line numbers. The elements in a structure are indexed from 1 to N; whenever an element is inserted or deleted, the index associated with each higher-indexed element in the same structure is incremented or decremented. The unique *current element* is that element whose index is stored in the *element-pointer* state variable. When a structure is opened with the SPH_openStructure call, the element pointer is set to N (pointing to the last element) or to 0 for an empty structure. The pointer is incremented when a new element is inserted after the current element, and is decremented when the current element is deleted. The pointer may also be set explicitly by the programmer using absolute and relative positioning commands:

```
void SPH_setElementPointer (int index);
void SPH_offsetElementPointer (int delta);      /* + for forward, - for backward */
```

Because the index of an element changes when a preceding element is added or deleted in its parent structure, using element indices to position the element pointer is liable to error. Thus, SPHIGS allows an application to place "landmark" elements, called *labels*, within a structure. A label element is given an integer identifier when it is generated:

```
void SPH_label (int id);
```

The application can move the element pointer via

```
void SPH_moveElementPointerToLabel (int id);
```

The pointer is then moved forward in search of the specified label. If the end of the structure is reached before the label is found, the search terminates unsuccessfully. Thus, it is advisable to move the pointer to the very front of the structure (index 0) before searching for a label.

Samsung/Google, EX1027, p. 340

7.9.2 Intrastructure Editing Operations

The most common editing action is insertion of new elements into a structure. Whenever an element-generating procedure is called, the new element is placed immediately after the current element, and the element pointer is incremented to point to the new element.[11]

Another form of insertion entails copying all the elements of a given structure into the open structure (immediately after the current element):

```
void SPH_copyStructure (int structureID);
```

Elements are deleted by the following procedures:

```
void SPH_deleteElement (void);
void SPH_deleteElementsInRange (int firstIndex, int secondIndex);
void SPH_deleteElementsBetweenLabels (int firstLabel, int secondLabel);
```

In all cases, after the deletion is made, the element pointer is moved to the element immediately preceding the ones deleted, and all survivors are renumbered. The first procedure deletes the current element. The second procedure deletes the elements lying between and including the two specified elements. The third procedure is similar, but does not delete the two label elements.

Note that these editing facilities all affect an entire element or a set of elements; there are no provisions for selective editing of data fields within an element. Thus, for example, when a single vertex needs to be updated the programmer must respecify the entire polyhedron.

An editing example. Let us look at a modification of our simple street example. Our street now consists of only the first house and the cottage, the former being fixed and the latter being movable. We create a label in front of the cottage, so we can subsequently edit the transformation in order to move the cottage.

To move the cottage, we reopen the street structure, move the pointer to the label, and then offset to the transformation element, replace the transformation element, and close the structure. The screen is automatically updated after the structure is closed, to show the cottage in its new position. This code is shown in Fig. 7.22(a), and its sequence of operations is illustrated in (b).

7.9.3 Instance Blocks for Editing Convenience

The previous editing example suggests that we place a label in front of each element we wish to edit, but creating so many labels is clearly too laborious. There are several techniques for avoiding this tedium. The first is to bracket an editable group of elements with two labels, and to use the labels in deleting or replacing the entire group. Another common technique is to group the set of elements in a fixed format and to introduce the group with a single label.

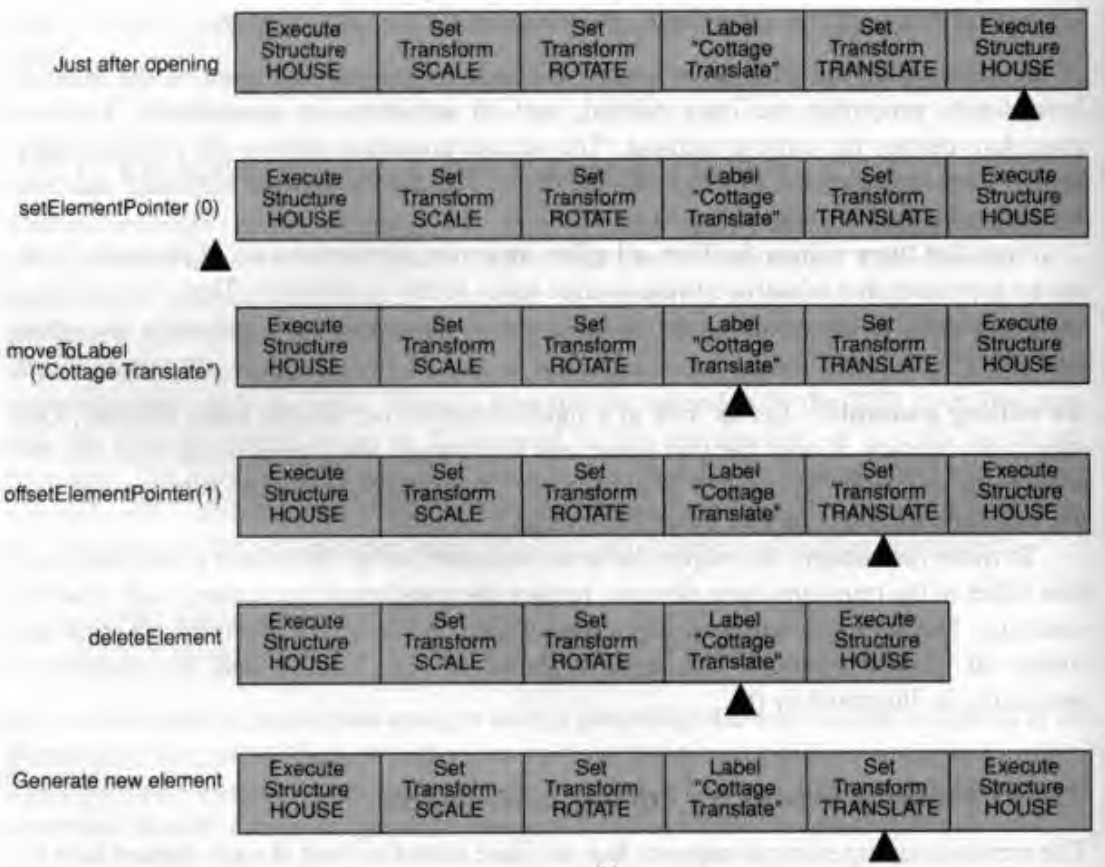
---

[11] We show the use of insert "mode" in our editing examples; however, SPHIGS also supports a "replace" editing mode in which new elements write over extant ones. See the reference manual for details.

Samsung/Google, EX1027, p. 341

```
SPH_openStructure (STREET_STRUCT);
    /* When a structure is opened, the element pointer is initially at its very end. We */
    /* must first move the pointer to the beginning, so we can search for labels. */
    SPH_setElementPointer (0);
    SPH_moveElementPointerToLabel (COTTAGE_TRANSLATION_LABEL);
    SPH_offsetElementPointer (1);        /* Pointer now points at transform element. */
    SPH_deleteElement ();                /* We replace here via a delete/insert combination */
    SPH_setLocalTransformation (newCottageTranslationMatrix, PRECONCATENATE);
SPH_closeStructure ();
```

(a)



(b)

Fig. 7.22 Editing operations. (a) Code performing editing. (b) Snapshot sequence of structure during editing. The black triangle shows the element pointer's position. (Syntax of calls abbreviated for illustrative purposes.)

Samsung/Google, EX1027, p. 342

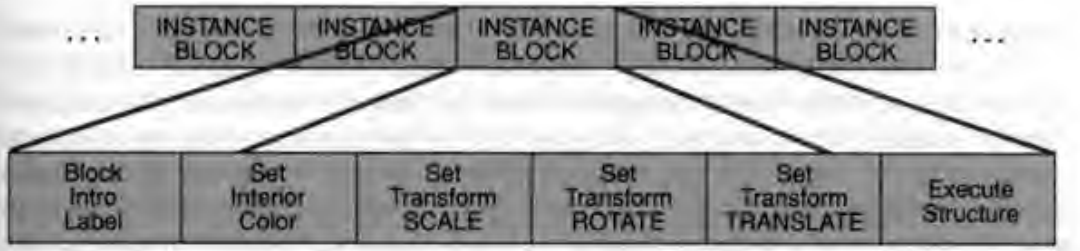


Fig. 7.23 Sample instance-block format.

To edit any member of the group, one moves the element pointer to the label, then offsets the pointer from that label into the group itself. Because the group's format is fixed, the offset is an easily determined small integer.

A special case of this technique is to design a standard way of instantiating substructures by preceding the structure-execution element with a common list of attribute-setting elements. A typical format of such a sequence of elements, called an *instance block*, is shown in Fig. 7.23; first comes the label uniquely identifying the entire block, then an interior-color setting, then the three basic transformations, and finally the invocation of the symbol structure.

We can create a set of symbolic constants to provide the offsets:

```
const int INTERIOR_COLOR_OFFSET = 1;
const int SCALE_OFFSET = 2;
const int ROTATION_OFFSET = 3;
const int TRANSLATION_OFFSET = 4;
```

Using the fixed format for the block guarantees that a particular attribute is modified in the same way for any instance. To change the rotation transformation of a particular instance, we use the following code:

```
SPH_openStructure (ID of structure to be edited);
    SPH_setElementPointer (0);
    SPH_moveElementPointerToLabel (the desired instance-block label);
    SPH_offsetElementPointer (ROTATION_OFFSET);
    SPH_deleteElement ();
    SPH_setLocalTransformation (newMatrix, mode);
SPH_closeStructure ();
```

Another nice feature of instance blocks is that the label introducing each block is easy to define: If the application keeps an internal database identifying all instances of objects, as is common, the label can be set to the unique number that the application itself uses to identify the instance internally.

7.9.4 Controlling Automatic Regeneration of the Screen Image

SPHIGS constantly updates the screen image to reflect the current status of its structure storage database and its view table. On occasion, however, we want to inhibit this regeneration, either to increase efficiency or to avoid presenting the user with a continuously

Samsung/Google, EX1027, p. 343

changing image that is confusing and that shows irrelevant intermediate stages of editing.

As we have seen, SPHIGS itself suppresses regeneration during the editing of a structure; no matter how many changes are made, an image is regenerated only when the structure is closed. This "batching" of updates is done for efficiency, since any deletion or transformation of a primitive can cause an arbitrary amount of damage to the screen image—damage that requires either selective damage repair or brute-force retraversal of all posted networks in one or more views. It is clearly faster for SPHIGS to calculate the cumulative effect of a number of consecutive edits just once, before regeneration.

A similar situation arises when several consecutive changes are made to different structures—for instance, when a structure and its substructures are deleted via consecutive calls to deleteStructure. To avoid this problem, an application can suppress automatic regeneration before making a series of changes, and allow it again afterward:

```
void SPH_setImplicitRegenerationMode (ALLOWED / SUPPRESSED value);
```

Even while implicit regeneration is suppressed, the application may explicitly demand a screen regeneration by calling

```
void SPH_regenerateScreen (void);
```

7.10 INTERACTION

Both SRGP's and SPHIGS's interaction modules are based on the PHIGS specification, and thus they have the same facilities for setting device modes and attributes, and for obtaining measures. The SPHIGS keyboard device is identical to that of SRGP, except that the echo origin is specified in NPC space with the z coordinate ignored. The SPHIGS locator device's measure has an additional field for the z coordinate, but is otherwise unchanged. SPHIGS also adds two new interaction facilities. The first is *pick correlation*, augmenting the locator functionality to provide identification of an object picked by the user. The second is the *choice* device, described in the reference manual, which supports menus. Section 10.1 provides a critical review of the PHIGS interaction devices in general.

7.10.1 Locator

The SPHIGS locator returns the cursor position in NPC coordinates, with $z_{NPC} = 0$.[12] It also returns the index of the highest-priority view whose viewport encloses the cursor.

```
typedef struct {
    point position;          /* [x, y, 0]NPC screen position */
    int viewIndex;           /* Index of view whose viewport encloses the cursor */
    int buttonOfMostRecentTransition;
    enum {UP, DOWN} buttonChord[MAX_BUTTON_COUNT];
} locatorMeasure;
```

[12] In PHIGS, the locator returns points in the 3D world-coordinate system. Many implementations, however, cannot return a meaningful z value; only high-performance workstations that support control dials and multiple real-time views can offer a comfortable user interface for pointing in 3D (see Chapter 8).

Samsung/Google, EX1027, p. 344

When two viewports overlap and the cursor position lies in the intersection of their bounds, the viewport having the highest index (in the view table) is returned in the second field. Thus, the view index is used to establish view priority for input as well as for output. The view-index field is useful for a variety of reasons. Consider an application that allows the user to specify bounds of a viewport interactively, much as one can move or resize a window manager's windows. In response to a prompt to resize, the user can pick any location within the viewport. The application program can then use the *viewIndex* field to determine which view was picked, rather than doing a point-in-rectangle test on viewport boundaries. The view index is also used in applications with some output-only views; such applications can examine the returned view index to determine whether the correlation procedure even needs to be called.

7.10.2 Pick Correlation

Because the SPHIGS programmer thinks in terms of modeled objects rather than of the pixels composing their images, it is useful for the application to be able to determine the identity of an object whose image a user has picked. The primary use of the locator, therefore, is to provide an NPC point for input to the pick-correlation procedure discussed in this section. As we saw with SRGP, pick correlation in a flat-earth world is a straightforward matter of detecting *hits*—primitives whose images lie close enough to the locator position to be considered chosen by the user. If there is more than one hit, due to overlapping primitives near the cursor, we disambiguate by choosing the one most recently drawn, since that is the one that lies "on top." Thus, a 2D pick correlator examines the primitives in inverse temporal order, and picks the first hit. Picking objects in a 3D, hierarchical world is a great deal more complex, for the reasons described next; fortunately, SPHIGS relieves an application of this task.

Picking in a hierarchy. Consider the complexity introduced by hierarchy. First, what information should be returned by the pick-correlation utility to identify the picked object? A structure ID is not enough, because it does not distinguish between multiple instances of a structure. Only the full *path*—a description of the complete ancestry from root to picked primitive—provides unique identification.

Second, when a particular primitive is picked, which level of the hierarchy did the user mean? For example, if the cursor is placed near one of our robot's thumbs, does the user mean to select the thumb, the arm, the upper body, or the entire robot? At times, the actual primitive is intended, at times the leaf structure is intended, and any other level is possibly intended, up to the very root! Some applications resolve this problem by providing a feedback mechanism allowing the user to step through the levels from primitive to root, in order to specify exactly which level is desired (see Exercise 7.13).

Comparison criterion. How is proximity to an object defined when the comparison should really be done in 3D? Since the locator device effectively yields a 2D NPC value, there is no basis for comparing the z coordinates of primitives to the locator position. Thus, SPHIGS can compare the cursor position only to the screen images of the primitives, not to the WC locations of the primitives. If a primitive is a hit, it is deemed a *candidate* for correlation. In wireframe mode, SPHIGS picks the very first candidate encountered during

Samsung/Google, EX1027, p. 345

traversal; the reason for this strategy is that there is no obvious depth information in a wireframe image, so the user does not expect pick correlation to take relative depth into account. (A side effect of the strategy is that it optimizes pick correlation.) In shaded rendering modes, SPHIGS picks the candidate whose *hit point* (the NPC point, on the primitive's normalized (3D NPC) surface, to which the user pointed directly) is closest to the viewpoint—the one "in front," as discussed in Section 7.12.2.

Pick-correlation utility. To perform pick correlation, the application program calls a SPHIGS pick-correlation utility[13] with an NPC point and a view index, typically returned by a previous interaction with the locator:

```
void SPH_pickCorrelate (
    point position, int viewIndex, pickInformation *pickInfo);
```

The returned information identifies the primitive picked and its ancestry via a *pick path*, as described by Pascal data types in Fig. 7.24.

When no primitive is close enough to the cursor position, the value of *pickLevel* returned is 0 and the *path* field is undefined. When *pickLevel* is greater than 0, it specifies the length of the path from the root to the picked primitive—that is, the primitive's depth within the network. In this latter case, entries [1] through [*pickLevel*] of the *path* array return the identification of the structure elements involved in the path leading from root to picked primitive. At the deepest level (entry [*pickLevel*]), the element identified is the primitive that was picked; at all other levels (entries [*pickLevel*–1] through [1]), the elements are all structure executions. Each entry in *path* identifies one element with a record that gives the structure ID of the structure containing the element, the index of the element

[13]Full PHIGS has the Pick logical input device that returns the same measure as the SPH_pickCorrelate procedure.

```
typedef struct {
    int structureID;
    int elementIndex;
    /* Enumerated type: polyline, polyhedron, execute-structure, etc. */
    elementTypeCode elementType;
    int pickID;
} pickPathItem;

typedef pickPathItem pickPath[MAX_HIERARCHY_LEVEL];

typedef struct {
    int pickLevel;
    pickPath path;
} pickInformation;
```

Fig. 7.24 Pick-path storage types.

Samsung/Google, EX1027, p. 346

in that structure, a code presenting the type of the element, and the pick ID of the element (discussed next).

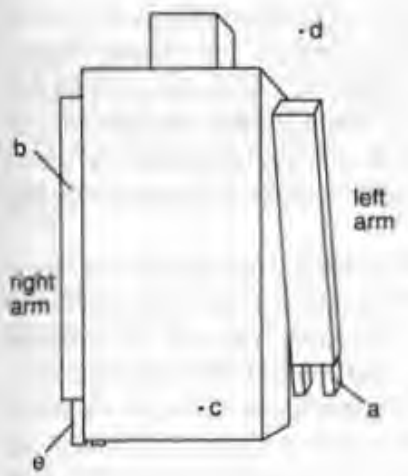
Figure 7.25 uses the structure network of Fig. 7.15 for the robot's upper body, and shows the pick information returned by several picks within the structure's displayed image.

How does the pick path uniquely identify each instance of a structure that is invoked arbitrarily many times within the hierarchy? For example, how do we distinguish a pick on the robot's left thumb from a pick on its right thumb? The pick paths for the two thumbs are identical except at the root level, as demonstrated by points *a* and *e* in Fig. 7.25.

The *pick identifier* can provide pick correlation at a finer resolution than does a structure ID. Although the element index can be used to identify individual elements, it is subject to change when the structure is edited. Therefore, using the pick ID is easier, because the pick ID is not affected by editing of other elements. It has a default value of 0 and is modally set within a structure. One generates a pick-ID element via

void SPH_setPickIdentifier (**int** *id*);

The pick-ID element is ignored during display traversal. Also, a pick ID has no notion of inheritance: it is initially 0 when SPHIGS begins the traversal of any structure, whether it is a root or a substructure. Because of these two aspects, pick IDs do not behave like attributes. Multiple primitives within a structure may have unique IDs or share the same one; this permits arbitrarily fine resolution of pick correlation within a structure, as needed by the application. Although labels and pick IDs are thus different mechanisms, the former used for editing and the latter for pick correlation, they are often used in conjunction. In particular, when structures are organized using the instance-block technique described in Section 7.9.2, a pick-ID element is also part of the block, and the pick ID itself is typically set to the same integer value as that of the block label.



Refer to the structure network shown in Fig. 7.15.

(a) level = 3
path[1] : struct UPPER _BODY, element 7
path[2] : struct ARM, element 3
path[3] : struct THUMB, element 1

(b) level = 2
path[1] : struct UPPER_BODY, element 11
path[2] : struct ARM, element 1

(c) level = 1
path[1] : struct UPPER_BODY, element 1

(d) level = 0

(e) level = 3
path[1] : struct UPPER_BODY, element 11
path[2] : struct ARM, element 3
path[3] : struct THUMB, element 1

Fig. 7.25 Example of pick correlation.

Samsung/Google, EX1027, p. 347

7.11 ADDITIONAL OUTPUT FEATURES

7.11.1 Attribute Bundles

Standard PHIGS provides a mechanism for setting attribute values indirectly. An application can, during its initialization sequence, store a collection of attribute values in an *attribute bundle*. Each type of primitive has its own type of bundle, and a PHIGS package provides storage for many bundles, each bundle identified by an integer ID. For example, we could store a "favorite" polyline attribute set in bundle 1. Subsequently, while editing a structure, we would prepare for the specification of a polyline primitive by inserting an element that, when executed during traversal, specifies that polyline attributes are to be taken from bundle 1 (rather than from the explicitly specified traversal attribute state).

Attribute bundles are often used as a "shorthand" to simplify the task of specifying attributes. Consider an application in which a large number of unrelated primitives must appear with identical attributes. Because the primitives are not related, the inheritance mechanism does not help. Indeed, without attribute bundles, the application would have to specify the desired attribute set redundantly, at various places throughout the structure networks.

Implementors of PHIGS packages sometimes initialize the attribute bundles in order to provide workstation-dependent preselected attribute sets that take advantage of the workstation's best features. The application programmer can choose to accept the bundles' initial values, as "suggestions" from the implementor, or to modify them with the bundle-editing commands. Changing definitions of bundles in the bundle table without changing structure networks is a simple mechanism for dynamically changing the appearance of objects.

7.11.2 Name Sets for Highlighting and Invisibility

SPHIGS supports two traditional feedback techniques that applications commonly use in conjunction with the SPHIGS picking facility: highlighting objects and making objects invisible. The former technique is typically used to provide feedback when the user picks an object; the latter declutters the screen by showing only desired detail. By default, all primitives that are part of a posted network are visible and unhighlighted. A set of primitives may be given an integer *name*, to identify the primitives for subsequent toggling of their visibility or highlighting status.

Because a group of unrelated primitives can share a name, and because a single primitive can have any number of names, the name feature can allow a complex object to be organized in several ways orthogonal to the structure hierarchy induced by structure invocation. For instance, an office-building model represented as a set of floor substructures could also be represented as a union of several systems: the plumbing network, the electrical wiring, and so on. Simply by giving all pipe primitives a common name (PLUMBING), we ensure that, even though the pipe primitives may be scattered among the actual structure hierarchy, we can nevertheless refer to them as a single unit.

When we want, say, to make the plumbing subsystem invisible, we add the name PLUMBING to the global *invisibility filter*; the screen is immediately updated to remove images of pipe objects. Similarly, by setting the invisibility filter to the names of all the

Samsung/Google, EX1027, p. 348

subsystems except the electrical subsystem, we can display the electrical subsystem in isolation. The highlighting filter works similarly. Both filters are initially empty, and are affected only by explicit calls that add or remove names.[14] Note that changing a filter, like changing a viewing specification, triggers screen regeneration; indeed, these operations change the rendered view of the CSS much as queries in traditional database programs are used to show different "views" of data.

The method used to bind names to primitives dynamically is very similar to the way attributes are assigned to primitives. SPHIGS maintains, as part of display traversal state, a *traversal name set* of zero or more names. A root inherits an empty name set. A child inherits the parent's name set when it is invoked, as it does attributes in general; thus, multiple instances of a building-block object can either share names or be named individually. The SPHIGS reference manual describes structure elements that, when executed during traversal, add names or remove names from this name set.

7.11.3 Picture Interchange and Metafiles

Although PHIGS and other standard graphics packages are system- and device-independent to promote portability, a given implementation of such packages in a particular environment is likely to be highly optimized in a nonportable way for performance reasons. The internal representation of the CSS, for example, may contain machine-specific information for structures and elements. To provide a medium of exchange among different implementations of PHIGS, the graphics standards committee has defined an archive file format. This portion of the standard is a machine- and environment-independent form of the contents of the CSS, without any viewing information. The PHIGS archive file thus is a portable snapshot of the structure database at a given time and permits PHIGS implementations to share geometric models.

PHIGS implementations may also support the writing of a metafile, which can contain a snapshot of what the application is currently presenting on the display surface. When these metafiles conform to the ANSI and ISO Computer Graphics Metafile (CGM) standard [ARNO88], the pictures contained in them can be transferred to such application environments as desktop publishing and interactive graphics art enhancement workstations. A CGM file is also a machine- and device-independent form of the CSS, but, unlike archive files, viewing information is also used in the creation of the picture represented in the CGM.

The CGM is typically created by having a PHIGS output device driver traverse the CSS to produce code for the CGM "virtual device," much as an ordinary device driver produces display code for a real display system. Other systems then can read the metafile into their CSS via an input device driver that converts from the prescribed metafile format to whatever is used by the particular implementation. Because the metafile is a 2D view of a 3D scene, any application obtaining the picture information via a CGM will have only the 2D view to work with: the original 3D geometry will be lost. If the 3D model is to be exchanged in a standard format, archive files must be used.

---

[14] The PHIGS detectability filter allows the application to specify primitives that cannot be picked. Moreover, PHIGS' filter scheme is more powerful, having separate inclusion and exclusion filters.

Samsung/Google, EX1027, p. 349

Other types of metafiles might be useful for debugging and backup purposes. An audit trail metafile is an historical transcript file containing a list of all calls to the graphics package procedures (and the values sent to them as parameters) in temporal order. It would, therefore, need to be run from start to finish, in order to recreate the CSS at the end of the session. Another type of transcript file records user actions: Running the application program with that transcript file reproduces the original sequence of PHIGS calls and thus the same CSS/image. No standards exist nor are standards planned for such transcript files, and neither a CGM nor a PHIGS archive file contains any historical information.

7.12 IMPLEMENTATION ISSUES

Much of the internal functionality of SPHIGS involves the maintenance (editing) and use (traversal) of the view table and the CSS. We do not discuss maintenance here, since it is essentially a conventional data-structures problem and not graphics-specific. Rather, this section focuses on the mechanics of displaying structures and of doing pick correlation.

The display traverser is invoked whenever the screen image must be updated. When implicit regeneration is allowed, the following operations prompt traversal: closing a structure, posting and unposting, changing a viewing transformation, changing rendering mode, and changing a filter. To generate the contents of a view, each structure posted to the view is traversed.

To display a posted structure network, SPHIGS visits the component structures' elements using a recursive descent, depth-first traversal, and performs the appropriate action for each element, based on the element's type. This display process that maps a model to an image on screen (or hardcopy) is referred to as display traversal in the context of PHIGS, but more generally as *rendering;* its implementation in software and/or hardware is referred to as the *rendering pipeline*.

Pick-correlation traversal is very similar to display traversal. The primitives encountered during traversal are compared to the locator position to find candidates. Traversal can be halted at the first candidate when wireframe rendering is being used; otherwise, a complete traversal is performed and the candidate closest to the viewpoint in z is chosen.

7.12.1 Rendering

The conceptual *rendering pipeline* that implements display traversal is illustrated in Fig. 7.26. Its first stage is the actual depth-first traversal of the CSS itself. (Alternatively, if an immediate-mode graphics package is used, the application may traverse the application model or generate primitives and attributes procedurally.) Each primitive encountered during traversal is passed through the remainder of the pipeline: First, the modeling transformations (described in Chapter 5) are applied to map the primitive from modeling coordinates to world coordinates. Then, the viewing operation is applied to transform and clip the primitive to the canonical view volume, and then to map it to the NPC parallelepiped (described in Chapter 6). Since these processes are independent of the display device and deal with vertex geometry in floating-point coordinates, this portion of

Samsung/Google, EX1027, p. 350

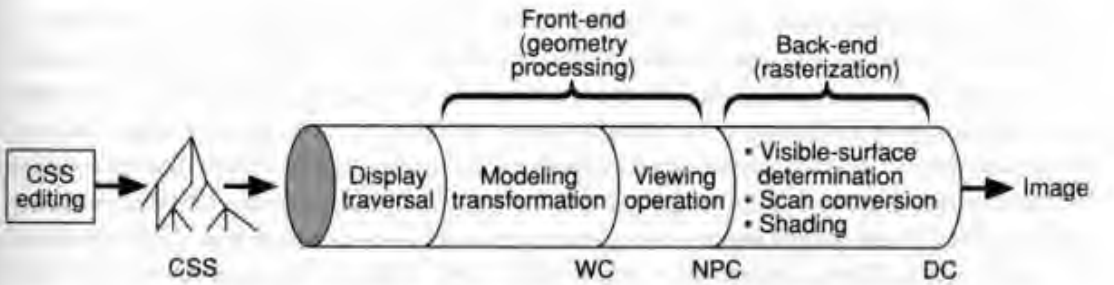


Fig. 7.26 The SPHIGS rendering pipeline.

the pipeline immediately following traversal is often referred to as the geometry-processing subsystem.

The back end of the pipeline takes transformed, clipped primitives and produces pixels; we will refer to this pixel processing as *rasterization*. This process is, of course, straightforward for wireframe mode: The NPC coordinates are easily mapped (via scaling and translating, with z ignored) to integer device coordinates, and then the underlying raster graphics package's line-drawing function is invoked to do the actual scan conversion. Shaded rendering, however, is quite complex, and is composed of three subprocesses: *visible-surface determination* (determining which portions of a primitive are actually visible from the synthetic camera's point of view), *scan conversion* (determining the pixels covered by a primitive's image), and *shading* (determining which color to assign to each covered pixel). The exact order in which these subprocesses are performed varies as a function of the rendering mode and implementation method. Detailed descriptions of the rasterization subprocesses are contained in Chapters 14 through 16.

Traversal. Since all stages of the rendering pipeline but the first one are covered in other chapters, we need to discuss only the traversal stage here. In a simple implementation, SPHIGS regenerates the screen by erasing it and then retraversing all posted roots (a list of which is stored with each view). Optimizing regeneration in order to traverse as little of the CSS as possible is quite difficult, because the effect of a trivial operation is potentially enormous. For example, it is difficult to determine how much of the screen must be regenerated due to the editing of a structure: It is possible that the structure is never invoked and thus has no effect on the screen, but it is also possible that it is a commonly invoked building block appearing in most of the views. Even when the implementation can determine that only a single view has been affected by an operation, refreshing that view may damage the images of objects in overlapping viewports of higher priority. Doing damage repair efficiently is, in general, a complicated task requiring considerable bookkeeping.

In implementations on high-performance workstations, where the image can be traversed and regenerated in a fraction of a second, the bookkeeping space and time overhead needed to regenerate the screen selectively is probably not worthwhile, and therefore complete regeneration is used most frequently. In either the complete or selective regeneration schemes, the double-buffering technique described later in this section can be

Samsung/Google, EX1027, p. 351

used to prevent annoying visual discontinuities on the screen during display traversal.

The traversal of a structure can be done in hardware or software; here, we show a procedural-language implementation, the pseudocode for which is given in Fig. 7.27. The procedure inherits a *traversal state*, which it then maintains as its local state for the duration of its activation. The attribute values (including the name set) and three transformation matrices (GM, LM, and their product, CMTM) form the traversal state, and the attributes and the CMTM are passed down from parent activation to child activation. (The activation of the procedure for a root structure inherits an identity GM and an attribute record filled with default values). The procedure visits each element in the structure and performs an operation based on the element's type, as indicated in Fig. 7.27.

Optimization via extent checking. The traversal strategy we have presented traverses a network's contents unconditionally; during traversal, all structure invocations are executed, and no part of the DAG is skipped. Frequently, however, not all of a network's objects are visible, since modeling and viewing tranformations in effect when the traversal is performed can cause large parts of a network to lie outside of the viewing volume.

What information do we need to trivially reject a subordinate structure during traversal? Say we have come to an element "Execute structure S" and can quickly compute the bounds of this instance, S_i, in NPC space. We could then ask the question, "Does S_i lie completely outside of the NPC viewport?" If it does, we skip over the structure-execution element and refrain from descending into structure S. Since structure S may be the root of

```
void TraverseStructureForDisplay (structID, attributeState, GM)
{
    LM = identity matrix;
    CMTM = GM;

    for (each element in structure structID) {
        switch (element type) {
            case attribute or name-set modification:
                update attributeState;
                break;
            case LM setting:
                replace or update LM;
                update CMTM by postconcatenating LM to GM;
                break;
            case primitive:
                pass primitive through rest of rendering pipeline;
                break;
            case execute structure:      /* A recursive call */
                TraverseStructureForDisplay
                    (ID of structure to be executed, attributeState, CMTM);
                break;
            default:                     /* ignore labels and pickIDs */
                break;
        }
    }
} /* TraverseStructureForDisplay */
```

Fig. 7.27 Pseudocode for display traversal.

Samsung/Google, EX1027, p. 352

an arbitrarily complex subnetwork, avoiding its traversal can reap potentially enormous time savings. In fact, this method allows us to trivially reject an entire network by comparing the root structure's NPC bounds with that of the viewport.

To implement this optimization, we need a simple method for computing the bounds of an arbitrarily complex object, and an efficient way of comparing these bounds to the NPC viewport. The representation that fits both these criteria is the *NPC extent*, defined as the smallest bounding box that completely encloses the NPC version of the object and is aligned with the principal axes. If we can show that the extent of an instance does not intersect with the viewport, we have proved that the instance is completely invisible. The extent is ideal because a viewport is itself an aligned box, and because it is very cheap to calculate the intersection of two aligned boxes (see Exercise 7.5). Trivial rejection is not the only optimization gained by the use of the NPC extent. We can also trivially accept—that is, discover instances that lie completely within the view volume and thus do not need to be clipped. The extent technique for trivial-accept/trivial-reject testing of substructures was first used in hardware in the BUGS system [VAND74] and is described also in [CLAR76].

Because instances are not explicitly stored in the CSS, we must calculate an instance's NPC extent from its structure's MC extent, which we store in the CSS. To perform extent checking during traversal, we thus first transform the MC extent to NPC, to determine the instance's NPC extent. Because a transformed extent box is not necessarily upright in NPC, we must determine the NPC extent of the transformed MC extent; that NPC extent is what we compare against the viewport.

To calculate the extent of a structure S, we must calculate the union of the extents of its primitives and descendants. We can calculate the extent of a polyhedron or fill area by traversing its MC vertex list, transformed by the local matrix, to determine minimum and maximum values of x, y, and z. These six numbers determine the extent: the box with one corner at $(x_{min}, y_{min}, z_{min})$ and the opposite corner at $(x_{max}, y_{max}, z_{max})$.

There is one other issue involved in maintaining extent information. When should the MC extent of a structure be calculated? It is not enough to do it whenever the structure is edited; a structure's MC extent is affected not only by its contents but also by the contents of any of its descendants. Thus, after each editing operation, an arbitrary number of structure extents must be recalculated. Moreover, recalculation requires traversal of an arbitrarily large subset of the CSS. We can optimize extent calculation by doing it not after each edit but during normal display traversal. This technique has the advantage of updating the extents of only those structures that are actually visible (part of a posted network), and it ensures that no structure's extent is calculated more than once in response to an arbitrarily large batch of editing operations. [SKLA90] presents this optimization technique in greater detail.

Animation and double-buffering. Software implementations of SPHIGS are well suited for rapidly producing animation prototypes, but are poorly suited for presenting high-quality real-time animations, because rendering is inherently time-consuming and the extent-checking optimization technique does not work well when a great deal of editing is performed between "frames." Animation prototypes are usually rendered in WIREFRAME mode, with automatic regeneration disabled, using a cyclical algorithm: The application

Samsung/Google, EX1027, p. 353

edits the structure database to describe the next scene, explicitly requests regeneration, then edits to describe the next scene, and so on. If the scene is not very complicated, WIREFRAME mode can be fast enough to present an almost–real-time animation of a modest number of primitives with just software. With some hardware assistance, higher-quality shading modes can perform in real-time.

One side effect of some simple implementations is that, between the "frames" of the animation, viewers see the screen being erased and can (more or less) see the objects being drawn as the traverser performs regeneration. An SPHIGS implementation can reduce this visual artifact by double buffering: using an offscreen canvas/bitmap to store the next frame while that frame is being drawn; then, when the regeneration is complete, the contents of that canvas can be copied onto the screen. When this technique is used on a system with a very fast copyPixel operation, the switch between frames is not noticeable, but there are still discontinuities in the motion of the objects in the scene. In fact, the *animation rate* (i.e., the number of frames per second) may decrease due to the increased overhead of the copyPixel, but the decrease in visual artifacts in many cases justifies the cost. Hardware double-buffering, as described in Section 4.4.1, is much better, since it avoids the copyPixel time.

7.12.2 Pick Correlation

In pick correlation, SPHIGS traverses those networks posted to the viewport in which the specified NPC point lies. The traversal is nearly identical to that performed during display; the modeling-transformation matrices are maintained, and much of the rendering pipeline is performed. Moreover, the pick ID, ignored during display traversal, is maintained as part of the local traversal state in the recursive traversal procedure. (The attribute group does not need to be maintained because during hit detection SPHIGS does not take into account attributes such as line thickness.)

Let us first examine pick correlation for wireframe rendering, where traversal is halted at the very first hit. Because traversal is recursive, the pick path is easily determined the moment the first hit is found: Each activation of the traversal procedure is responsible for one level of information in the pick path. Each primitive is transformed to NPC before being compared to the locator position for hit detection. (Hit-detection techniques are described later in this section.) When a hit is discovered by an activation of the procedure, it returns, and the recursion is unwound. Before returning, each activation stores its pick information into one level of a global pick-information array. (See Exercise 7.9.)

In shaded rendering modes, the order in which primitives are encountered during traversal has no bearing on whether they appear in front of or in back of other primitives whose images map to the same part of the screen. Therefore, the SPHIGS pick-correlation algorithm cannot simply choose the first hit detected. Rather, it must traverse all posted structure networks, maintaining a list of candidate hits. When traversal is completed, the candidates are compared to determine the candidate whose NPC hit point is closest to the viewpoint—that is, the one whose z coordinate is algebraically largest. To calculate a candidate's z coordinate at the hit point, we can plug the x and y coordinates of the locator measure into the appropriate equations for each primitive: the (parametric) line equation for edges, and the plane equation for facets (see Exercises 7.7 and 7.8). Another approach

Samsung/Google, EX1027, p. 354

using a hardware visible-surface determination algorithm is described in Section 15.4. Pick-correlation traversal can, of course, be optimized in a variety of ways, including the extent-checking techniques used to optimize display traversal.

Analytical hit detection. Two fundamental methods, analytical and clipping, are used for hit detection. For analytical hit detection, algebraic equations are used to determine whether the NPC primitive lies sufficiently close to the 2D NPC locator measure. We first convert to 2D by ignoring the z coordinate for an orthographic projection or use the perspective transform of Section 6.5.4 to create an orthographic view volume. Some examples of analytical techniques follow:

- In WIREFRAME mode, a function PtNearLineSegment is used to compute the distance in NPC from the cursor position to each edge of a facet or fill area, and to each line segment of a polyline. This same function would be used in shaded rendering modes for polyline primitives. The line equation is used for the computation (see Exercise 7.10).
- In shaded rendering modes, a function PtInPolygon is used to test for hits on fill areas and facets. One popular algorithm for determining if the NPC cursor position lies inside a polygon, based on the odd-parity rule described in Section 2.1.3, casts a ray from the locator position and determines how many times the ray intersects the polygon. The algorithm traverses the edge list, testing for intersections and special cases (e.g., intersections at vertices, edge–ray colinearity). The polygon scan-conversion algorithm described in Section 3.7 tackles a very similar problem and can be adapted for use as a PtInPolygon function (see Exercise 7.12). This algorithm handles the general case of concave and self-intersecting polygons. Optimized computational geometry algorithms are available if it can be guaranteed that polygons do not self-intersect, or that a horizontal ray intersects only two edges, or that the polygon is convex [PREP85].
- Hit detection for nongeometric text is most easily performed by comparing the locator position to the text's rectangular screen extent.
- Packages that support primitives such as ellipses and curves and surfaces require more complex pick-correlation algorithms of the sort mentioned in Chapters 11 and 19. Furthermore, the problems are similar to those encountered in ray tracing, as described in Chapter 15.

Hit detection via clipping. Some hardware clipping devices and optimized software clipping utilities return state information, allowing an application to determine whether any part of a primitive's image lies inside a 2D integer clip rectangle, without having actually to draw the primitive. An SPHIGS implementation can use this type of clipping to test for candidacy: The clip rectangle is set to a *pick window*—a small square surrounding the cursor position—and then traversal is performed. Each primitive (transformed to integer device coordinates) is given to the clipper, which returns a Boolean "hit-detection" result (see Exercise 7.11). Alternatively, if the clipper does not return any such state information, we can draw each primitive into an offscreen bitmap using the pick-window clip rectangle; if any pixels are changed, the primitive is deemed a candidate.

Samsung/Google, EX1027, p. 355

7.13 OPTIMIZING DISPLAY OF HIERARCHICAL MODELS

7.13.1 Elision

We can model a building as a parts hierarchy by saying that it consists of floors, the floors consist of offices, and so on; there are no primitives in the hierarchy's nodes until we get to the level of bricks, planks, and concrete slabs that consist of polyhedra. Although this representation might be useful for construction purposes, it is not as useful for display, where we sometimes wish to see a cruder, simplified picture that eliminates confusing and unneeded details (and that is faster to render). The term *elision* refers to the decision by a display traverser to refrain from descending into a substructure.

Pruning. In Section 7.12.1, we showed how a display traverser can avoid executing a subordinate structure by checking its NPC extent against the viewport, to determine whether the substructure lies wholly outside (i.e., is fully clipped by) the view volume. This type of elision, a typical feature of optimized display traversers, is called *pruning*.

Culling. In addition, a traverser can examine the size of the subordinate's NPC extent, and can choose to elide the substructure if that substructure's extent is so small that, after transformation and projection, the image of the object would be compressed into a few pixels. This type of elision is called *culling*; on systems that support it, the application typically can specify a minimum extent size, below which a substructure is culled.

When a substructure is pruned, it is not drawn at all; however, that is not the best approach for culling. The object is being culled not because it is invisible from the current point of view, but because its image is too tiny for its details to be discernible. Rather than not draw it at all, most implementations draw it as an abstract form, typically a parallelipiped representing its WC extent, or simply a rectangle (the 2D projection of its NPC extent).

Level-of-detail elision. Pruning and culling are optimization techniques, preventing traversal that is unnecessary (pruning) or that would produce an unusable image (culling). Elision can also be used to give the user control over the amount of detail presented in a view of the CSS. For example, a user examining our building model could specify a low level of detail in order to view the building as simply a set of parallelepipeds representing the individual floors, or could increase the level of detail so as to see in addition the walls that form office boundaries.

The MIDAS microprocessor architecture simulator [GURW81] was one of the earliest systems in which alternate representations of subobjects were traversed automatically by the display processor, depending on the size of the projection of the subobject on the screen. This logical-zoom facility made successively more detail appear as the user dynamically zoomed in on the processor block in the CPU architecture block diagram. Also, at increased zoom factors, one could see digits representing address, data, and control bytes moving from source to destination over system buses during the simulation of the instruction cycle.

Elision in MIDAS was implemented using a conditional-execution facility of the BUGS vector-graphics system [VAND74]: The hierarchical display list included a conditional substructure execution element that tested the screen extent of the substructure. A similar

Samsung/Google, EX1027, p. 356

feature described in the 1988 specification of PHIGS+ in the form of a conditional-execution element, allowing pruning and culling to be performed explicitly by elements within the CSS.

7.13.2 Structure Referral

Certain implementations of PHIGS and PHIGS+ allow a nonstandard form of structure execution, called *referral*, that bypasses the expensive state saving and restoring of the ExecuteStructure mechanism. Whereas we could argue that a better, transparent approach to increasing efficiency is to optimize the implementation of the ExecuteStructure operation itself (so that the operation saves only as much state as required at any given time), it is simpler to add a ReferStructure operation and to let the programmer use it for those cases where an invoked child structure does not have any attribute-setting elements.

A second use of ReferStructure is to allow a child to influence its parents' attributes. This is useful when a group of objects do not have a common parent, but do need to "inherit" the same group of attributes, including potentially an arbitrary number of modeling transformations. In this case, we can create a structure *A* that consists of transformation and appearance attribute settings, and then have each of the object structures refer to structure *A*. By editing structure *A* later, we indirectly affect all structures referring to *A*. If only appearance attributes need to be affected, PHIGS attribute bundles provide an alternative, standard mechanism for changing the appearance of diverse structures.

7.14 LIMITATIONS OF HIERARCHICAL MODELING IN PHIGS

Although this chapter has emphasized geometric modeling hierarchy, it is important to realize that hierarchy is only one of many forms of data representation. In this section, we discuss the limitations of hierarchy in general and in PHIGS in particular; in the next section, we present some alternatives to structure hierarchy.

7.14.1 Limitations of Simple Hierarchy

As mentioned in Section 1.7.2, some applications have no real structure for their data (e.g., data for scatter plots), or have at most a (partial) ordering in their data (e.g., a function represented algebraically). Many other applications are more naturally expressed as networks—that is, as general (directed) graphs (which may have hierarchial subnets). Among these are circuit and electrical-wiring diagrams, transportation and communications networks, and chemical-plant–piping diagrams. Another example of simple hierarchy's insufficiency for certain types of models is Rubik's cube, a collection of components in which the network and any hierarchy (say of layers, rows, and columns) is fundamentally altered after any transformation.

For other types of models, a single hierarchy does not suffice. For example, the pen holder on an (x, y) plotter is moved by, and therefore "belongs" to, both the horizontal and vertical arms. In short, whether the application model exhibits pure hierarchy, pure network without hierarchy, hierarchy in a network with cross-links, or multiple hierarchies, SPHIGS can be used to display it, but we may not want, or be able, to use structure hierarchy in its full generality.

Samsung/Google, EX1027, p. 357

7.14.2 Limitations of SPHIGS "Parameter Passing"

The black-box nature of structures is good for modularity but, as shown in our robot example, can be limiting. For example, how can we build a robot with two identical arms and have the robot use its right arm to pick up a cylinder from a table and move away with that cylinder? The pick-up operation can be performed by editing the arm structure to add an invocation of the cylinder structure, so that as the robot or arm moves subsequently, the cylinder moves along with it. But if we implement the robot by having a single arm structure invoked twice by the upper-body structure, the result of that operation would be that both the left and right arm would be holding a cylinder! Thus, we have no choice but to build two separate arm structures, with unique structure IDs, each invoked only once. Let us extend this example. If we wish to have an army of robots with independently controlled arms, we must build two unique arm structures for each robot! Obviously, here is a case where structure hierarchy is not as useful as we first thought.

The reason that structure hierarchy does not support instances of structures differing in the settings of transformations at various hierarchical levels is that structure hierarchy has neither the general parameter-passing mechanism of procedure hierarchy nor general flow-of-control constructs. Rather, it is essentially a data organization with rudimentary interactions between structures and at most limited conditional execution of structures (in PHIGS+). We have seen that a parent's transformations are inherited by all the children, and there is no provision for a particular child to be affected selectively.

By contrast, in a procedure hierarchy, a "root" procedure passes parameters that are either used directly by procedures the root calls, or passed down by those procedures to lower-level procedures. Thus, the root can pass down parameters arbitrarily deeply and selectively via intermediate procedures. Furthermore, with parameters, a procedure can control not just the data on which a lower-level procedure operates, but even the way in which the lower-level procedure operates. To change its operation, the lower-level procedure uses flow-of-control constructs to test parameters and selectively enables or disables code segments. Because of structure hierarchy's lack of general parameter-passing and flow-of-control mechanisms, our analogy between structure hierarchy and procedure hierarchy in the introduction was a superficial one.

By augmenting the attribute-binding model of PHIGS, we could specify attributes for selected object instances at arbitrary levels in a hierarchy. A system that has such a general mechanism is SCEFO [STRA88], which allows the programmer to specify an attribute that is to be applied when the traversal reaches a certain state, the state being represented by a pathname similar to the pick path returned by the PHIGS pick device. With this facility, it would be possible to control individually the colors or positions of the thumb instances in our robot, without having to create two virtually identical arm masters, by making use of the fact that the two thumb instances have unique pathnames.

Another limitation in the PHIGS parameter-passing mechanism is that it handles transformations and appearance attributes for which inheritance rules are very simple. It would not be easy to support operations more complex than geometric transformations; a more general model is needed to support set operations on solid primitives (Chapter 12), and deformation operations such as bend, taper, and twist (Chapter 20).

Samsung/Google, EX1027, p. 358

7.15 ALTERNATIVE FORMS OF HIERARCHICAL MODELING

We have just concluded that structure hierarchy is only one way—and not always the best way—to encode hierarchy. In this section, we discuss alternatives to structure hierarchy.

7.15.1 Procedure Hierarchy

In the spectrum from pure data hierarchy to pure procedure hierarchy, structure hierarchy is almost all the way at the data end, since it lacks general flow of control. By contrast, a template procedure (i.e., a procedure defining a template object, consisting of primitives or of calls to subordinate template procedures) can use parameters and arbitrary flow of control. Template procedures can be used in two different ways. First, they can be used with a retained-mode graphics package such as SPHIGS. Here, they are used as a means to the end of creating structure hierarchy. Second, they can be used to specify primitives and attributes to an immediate-mode graphics package. Here, they are not means to ends, but rather are ends in themselves; that is, they are the sole representation of the hierarchy used for display. In this case, display traversal is effected by procedure traversal—practical only if the CPU can provide a reasonable rate of retraversal. (In general, smooth dynamics requires at least 15 frames per second; 30 frames per second looks noticeably better.) The procedures themselves implement inheritance and maintain transformation matrices and attribute states, using techniques similar to those presented in our discussion of traversal implementation in Section 7.12. Newman's display procedures mentioned in Section 7.4 were an early example of the use of procedure hierarchy for dynamics.

Pick correlation is a bit tricky in dynamic procedure traversal, since it requires retraversing the procedure hierarchy. If the first candidate is to be chosen (i.e., if the rendering mode is wireframe), it is difficult to halt the traversal as soon as a hit is detected and to return from an arbitrary level of the procedure activation stack. If a nonwire frame rendering mode is used, there must be a way of interacting with the rendering pipeline to obtain the candidates and their z values corresponding to the cursor position. Each procedure is also complicated by the fact that it must be used both for display and for correlation traversal.

We can combine procedure and structure hierarchy by using template procedures to create structures. For example, our robot can be built using template procedures for each of the parts; each procedure creates display commands or a SPHIGS structure, depending on whether we choose immediate mode or retained structure mode, respectively. The upper-body procedure can pass parameters to the arm and thumb procedures to initialize their transformations individually and to allow the arms and thumbs to operate independently. In retained mode, each invocation of the robot template procedure thus creates a unique SPHIGS network, with unique names for all the child structures. For example, two arm structures would be created, with their own, separate invocations of thumbs. Leaf nodes such as the thumb can still be shared among multiple networks, since they can be instanced with individual transforms by their callers. To create unique structure IDs, we can assign a unique interval of integer space to each root, and can number its parts within that interval.

Samsung/Google, EX1027, p. 359

What are the limitations of using only procedure hierarchy? First, unless the programming environment supports dynamic code creation, it is difficult to edit or construct procedures at run time. Creating new data structures is far easier. Therefore, procedure hierarchy is typically used when the set of template objects is predefined and only the attributes need to be varied dynamically. Structures created by template procedures can, of course, be edited with all the PHIGS machinery.

Second, if the procedure hierarchy is used to create immediate-mode graphics, the CPU, involved in constantly retraversing the procedure hierarchy, is much less available for application processing. We can offload the CPU in a multiprocessor system by having another CPU dedicated to procedure traversal. Alternatively, in retained mode, structure-network traversal can be performed by a special-purpose hardware coprocessor or a separate CPU.

Third, even for minor edits, an immediate-mode procedure hierarchy must be retraversed in its entirety and all display primitives must be retransmitted to the graphics packages. For display systems connected over networks or communication lines, this requirement produces heavy communication traffic. It is far more efficient for minor edits, for both the CPU and the communications system, to store and manage a structure database in a display peripheral that can be modified incrementally and traversed rapidly.

7.15.2 Data Hierarchy

Unlike procedure hierarchy, data hierarchy is well suited to dynamic creation. Like template-procedure hierarchy, it can be used in conjunction with either immediate- or retained-mode graphics packages. If immediate mode is used, the CPU must retraverse the application model and drive the package sufficiently quickly to provide dynamic update rates. Objects are created and edited by changing the application model and retraversing it to update the screen. The application must do its own pick correlation, by retraversal. As with an immediate-mode procedure hierarchy, however, if the display subsystem is on a communications network, retransmittal of graphics commands for any update is considerably slower than is sending update commands to a structure database in the peripheral.

Like the structure-hierarchy technique, the data-hierarchy approach lacks the flexibility of the procedure-hierarchy method due to the absence of general flow-of-control mechanisms; these must be embodied via flags in the data structures. Object-oriented environments with run-time code creation and binding offer an attractive, totally general combination of data and procedure hierarchy; there is a natural match between the notion of an object–subobject hierarchy and that of a class–instance hierarchy. As processor performance improves, object-oriented environments are likely to become a dominant paradigm in dynamic computer graphics.

7.15.3 Using Database Systems

Since a general-purpose database has more power than does a special-purpose one, we should consider using standard database systems for computer graphics [WELL76; GARR80]. Unfortunately, such databases are designed to work with large volumes of data in secondary storage and to give response times measured on a human time scale. They are

Samsung/Google, EX1027, p. 360

designed to process user-input queries or even batch transactions with times measured, at best, in milliseconds, whereas real-time graphics demands microsecond access to elements. Using a memory-resident database would work best if the database were optimized for fast traversal and had built-in graphics data types and operators. At least, it would have to be able to invoke procedures for retrieved items, passing parameters extracted from fields in the database.

Although several systems have used relational databases for graphics, the limitations on the structure of the data imposed by the relational model, as well as the slowness of standard relational databases, have restricted these systems to research environments. As object-oriented environments are useful for combining data and procedures, such an environment used in conjunction with an object-oriented database has the potential for removing the restrictions of relational databases. The slow performance of object-oriented databases, however, may make them infeasible for real-time graphics in the near term.

7.16 SUMMARY

This chapter has given a general introduction to geometric models, emphasizing hierarchical models that represent parts assemblies. Although many types of data and objects are not hierarchical, most human-made objects are at least partly so. PHIGS and our adaptation, SPHIGS, are designed to provide efficient and natural representations of geometric objects stored essentially as hierarchies of polygons and polyhedra. Because these packages store an internal database of objects, a programmer can make small changes in the database with little effort, and the package automatically produces an updated view. Thus, the application program builds and edits the database, typically in response to user input, and the package is responsible for producing specified views of the database. These views use a variety of rendering techniques to provide quality–speed tradeoffs. The package also provides locator and choice input devices, as well as pick correlation to allow the selection of objects at any level in a hierarchy. Highlighting and visibility filters can be used for selective enabling and disabling as another form of control over the appearance of objects.

Because the nature of structures and the means for searching and editing them are restricted, such a special-purpose system is best suited to motion dynamics and light update dynamics, especially if the structure database can be maintained in a display terminal optimized to be a PHIGS peripheral. If much of the structure database must be updated between successive images, or if the application database can be traversed rapidly and there is no bottleneck between the computer and the display subsystem, it is more efficient to use a graphics package in immediate mode, without retaining information.

Structure hierarchy lies between pure data and pure procedure hierarchy. It has the advantage of dynamic editing that is characteristic of data hierarchy. It also allows a simple form of parameter passing to substructures (of geometric or appearance attributes), using the attribute-traversal state mechanism. Because of the lack of general flow-of-control constructs, however, the parameter-passing mechanism is restricted, and structures cannot selectively set different attributes in different instances of a substructure. Instead, template procedures can be used to set up multiple copies of (hierarchical) structures that are identical in structure but that differ in the geometric or appearance attributes of substructures. Alternatively, they can be used to drive an immediate-mode package.

Samsung/Google, EX1027, p. 361

SPHIGS is oriented toward geometric models made essentially from polygons and polyhedra, especially those that exhibit hierarchy; in Chapters 11 and 12, we look at geometric models that have more complex primitives and combinations of primitives. Before turning to those more advanced modeling topics, we first consider interaction tools, techniques, and user interfaces.

EXERCISES

7.1 a. Complete the robot model of Fig. 7.16 by adding a base on which the upper body swivels and create a simple animation of its movement through a room.

b. Create an SPHIGS application producing an animation in which a one-armed robot approaches a table on which an object lies, picks up the object, and walks off with it (see Section 7.14 for the reason for specifying a one-armed robot).

7.2 Enhance the robot animation to provide user interaction. Let there be a number of objects on the table, and allow the user to choose (using the locator) the object that the robot should pick up.

7.3 Redesign the two-armed robot model so as to allow the thumbs on each arm to be controlled individually, so that each arm can pick up objects individually.

7.4 Enhance a robot animation so that three views of the animation are shown simultaneously, including one overhead orthographic view and one "robot's eye" view that shows us what the robot itself "sees" as it moves.

7.5 Design the addition of pruning elision to the recursive display traverser of Fig. 7.27. Assume the MC extent of a structure is stored in the structure's record. You must transform an MC extent box into an NPC extent box, meeting the requirement that extent boxes be aligned with the principal axes.

7.6 Update our recursive display traverser so that it maintains the MC extent information stored for each structure. Assume that, whenever a structure S is closed after being edited, a Boolean "extentObsolete" field in S's record is set. Assume also that functions are available that, given any primitive, return the primitive's NPC extent.

7.7 Design an algorithm for calculating analytically the hit point of a candidate line, given the line's NPC endpoints and the locator measure.

7.8 Design an algorithm for calculating analytically the hit point of a candidate fill area.

7.9 Design, using pseudocode, a recursive pick-correlation traverser that supports only wireframe mode.

7.10 Implement the function PtNearLineSegment analytically for use in pick correlation. To be a candidate, the line segment's image must come within P pixel widths of the locator position.

7.11 Implement the function PtNearLineSegment using clipping. Modify the Liang–Barsky clipping algorithm (of Fig. 3.45) to optimize it, because the clipped version of the segment is not needed—only a Boolean value is to be returned.

7.12 Implement the function PtInPolygon for use in pick correlation. Treat the special cases of rays that pass through vertices or are coincident with edges. See [PREP85] and [FORR85] for discussions of the subtleties of this problem.

7.13 Design a user interface for picking that lets the user indicate the desired level of a hierarchy. Implement and test your interface with the robot model by writing an application that allows the user to highlight portions of the robot's anatomy, from individual parts to whole subsystems.

Samsung/Google, EX1027, p. 362

8
Input Devices, Interaction Techniques, and Interaction Tasks

This is the first of three chapters on designing and implementing graphical user–computer interfaces. High-quality user interfaces are in many ways the "last frontier" in providing computing to a wide variety of users, since hardware and software costs are now low enough to bring significant computing capability to our offices and homes. Just as software engineering has recently given structure to an activity that once was totally ad hoc, so too the new area of user-interface engineering is generating user-interface principles and design methodologies.

Interest in the quality of user–computer interfaces is new in the formal study of computers. The emphasis until the early 1980s was on optimizing two scarce hardware resources, computer time and memory. Program efficiency was the highest goal. With today's plummeting hardware costs and increasingly powerful graphics-oriented personal-computing environments (as discussed in Chapter 1), however, we can afford to optimize user efficiency rather than computer efficiency. Thus, although many of the ideas presented in this chapter require additional CPU cycles and memory space, the potential rewards in user productivity and satisfaction well outweigh the modest additional cost of these resources.

The quality of the user interface often determines whether users enjoy or despise a system, whether the designers of the system are praised or damned, whether a system succeeds or fails in the market. Indeed, in such critical applications as air-traffic control and nuclear-power-plant monitoring, a poor user interface can contribute to and even cause accidents of catastrophic proportions.

The desktop user-interface metaphor, with its windows, icons, and pull-down menus, all making heavy use of raster graphics, is popular because it is easy to learn and requires

Samsung/Google, EX1027, p. 363

little typing skill. Most users of such systems are not computer programmers and have little sympathy for the old-style hard-to-learn keyboard-oriented command-language interfaces that many programmers take for granted. The designer of an interactive graphics application must be sensitive to users' desire for easy-to-learn yet powerful interfaces.

On the other hand, in the future, the level of computer sophistication of workers will increase, as more users enter the workforce already computer-literate through computer use at home and school. Developers of some educational and game software will continue to design for the computer-naive user, while developers of workplace systems will be able to assume an increased awareness of general computer concepts.

In this chapter, we discuss basic elements of user interfaces: input devices, interaction techniques, and interaction tasks. Input devices were introduced in Chapters 2 and 4; here we elaborate on their use. Interaction techniques are ways to use input devices to enter information into the computer, whereas interaction tasks classify the fundamental types of information entered with the interaction techniques. Interaction techniques are the primitive building blocks from which a user interface is crafted.

In Chapter 9, we discuss the issues involved in putting together the building blocks into a complete user-interface design. The emphasis is on a top-down design approach; first, design objectives are identified, and the design is then developed through a stepwise refinement process. The pros and cons of various dialogue styles—such as what you see is what you get (WYSIWYG), command language, and direct manipulation—are discussed, and window-manager issues that affect the user interface are also described. Design guidelines, the dos and don'ts of interface design, are described and illustrated with various positive and negative examples. Many of the topics in Chapters 8 and 9 are discussed in much greater depth elsewhere; see the texts by Baecker and Buxton [BAEC87], Hutchins, Hollan, and Norman [HUTC86], Mayhew [MAYH90], Norman [NORM88], Rubenstein and Hersh [RUBE84], and Shneiderman [SHNE87]; the reference book by Salvendy [SALV87]; and the survey by Foley, Wallace, and Chan [FOLE84].

Many of the examples used in Chapters 8 and 9 are taken from the user interface of Apple Computer's Macintosh. Although the Macintosh user interface is not perfect, it is a huge improvement over previous commonly available interfaces. Developed in the early 1980s, the Macintosh was built primarily on pioneering work at Xerox Corporation's Palo Alto Research Center (PARC) in the mid-1970s, and has been imitated and in some cases extended by systems such as Microsoft Windows, Presentation Manager, NeXT's NeXT Step, the Commodore Amiga, Digital Research's GEM, and many others. (Much of this book was written on the Macintosh, using Microsoft Word, and many of the figures were drawn on the Macintosh using Freehand.)

Chapter 10 treats user-interface software. It is one thing to design graphic user interfaces that are easy to learn and fast to use; it is quite another to implement them. Having the right software tools is of critical importance. This chapter reviews the input-handling capabilities of SRGP and SPHIGS, and then discusses more general and more powerful input-handling capabilities. The internal structures and implementation strategies of window managers, a critical element in many high-quality user interfaces, are described. Finally, the key concepts of user-interface management systems (UIMSs) are presented. UIMSs provide a means for interface designers and implementors quickly to develop, try out, and modify their interface concepts, and thus decrease the cost of the

Samsung/Google, EX1027, p. 364

essential testing and refinement steps in user-interface development.

We focus in this chapter on input devices—those pieces of hardware by which a user enters information into a computer system. We have already discussed many such devices in Chapter 4. In this chapter, we introduce additional devices, and discuss reasons for preferring one device over another. In Section 8.1.6, we describe input devices oriented specifically toward 3D interaction. We continue to use the logical device categories of locator, keyboard, choice, valuator, and pick used by SRGP, SPHIGS, and other device-independent graphics subroutine packages.

An *interaction task* is the entry of a unit of information by the user. The four basic interaction tasks are *position*, *text*, *select*, and *quantify*. The unit of information input in a position interaction task is of course a position. Similarly, the text task yields a text string; the select task yields an object identification; and the quantify task yields a numeric value. Many different *interaction techniques* can be used for a given interaction task. For instance, a selection task can be carried out by using a mouse to select items from a menu, using a keyboard to enter the name of the selection, pressing a function key, or using a speech recognizer. Similarly, a single device can be used for different tasks: A mouse is often used for both positioning and selecting.

Interaction tasks are distinct from the logical input devices discussed in earlier chapters. Interaction tasks are defined by *what* the user accomplishes, whereas logical input devices categorize *how* that task is accomplished by the application program and the graphics package. Interaction tasks are user-centered, whereas logical input devices are a programmer and graphics-package concept.

By analogy with a natural language, single actions with input devices are similar to the individual letters of the alphabet from which words are formed. The sequence of input-device actions that makes up an interaction technique is analogous to the sequence of letters that makes up a word. A word is a unit of meaning; just as several interaction techniques can be used to carry out the same interaction task, so too words that are synonyms convey the same meaning. All the units of meaning entered by the user can be categorized as one of the four basic interaction tasks, just as words can be categorized as verb, noun, adjective, and so on. An interactive dialogue is made up of interaction-task sequences, just as a sentence is constructed from word sequences.

8.1 INTERACTION HARDWARE

Here, we introduce some interaction devices not covered in Section 4.6, elaborate on how they work, and discuss the advantages and disadvantages of various devices. The presentation is organized around the logical-device categorization of Section 4.6, and can be thought of as a more detailed continuation of that section.

The advantages and disadvantages of various interaction devices can be discussed on three levels: device, task, and dialogue (i.e., sequence of several interaction tasks). The *device level* centers on the hardware characteristics per se, and does not deal with aspects of the device's use controlled by software. At the device level, for example, we note that one mouse shape may be more comfortable to hold than another, and that a data tablet takes up more space than a joystick.

Samsung/Google, EX1027, p. 365

At the *task level*, we might compare interaction techniques using different devices for the same task. Thus, we might assert that experienced users can often enter commands more quickly via function keys or a keyboard than via menu selection, or that users can pick displayed objects more quickly using a mouse than they can using a joystick or cursor control keys.

At the *dialogue level*, we consider not just individual interaction tasks, but also sequences of such tasks. Hand movements between devices take time: Although the positioning task is generally faster with a mouse than with cursor-control keys, cursor-control keys may be faster than a mouse *if* the user's hands are already on the keyboard and will need to be on the keyboard for the next task in sequence after the cursor is repositioned. Dialogue-level issues are discussed in Chapter 9, where we deal with constructing complete user interfaces from the building blocks introduced in this chapter. Much confusion can be avoided when we think about devices if we keep these three levels in mind.

Important considerations at the device level, discussed in this section, are the device footprints (the *footprint* of a piece of equipment is the work area it occupies), operator fatigue, and device resolution. Other important device issues—such as cost, reliability, and maintainability—change too quickly with technological innovation to be discussed here. Also omitted are the details of connecting devices to computers; by far the most common means is the serial asynchronous RS-232 terminal interface, generally making interfacing quite simple.

8.1.1 Locator Devices

It is useful to classify locator devices according to three independent characteristics: absolute or relative, direct or indirect, and discrete or continuous.

Absolute devices, such as a data tablet or touch panel, have a frame of reference, or origin, and report positions with respect to that origin. *Relative* devices—such as mice, trackballs, and velocity-control joysticks—have no absolute origin and report only changes from their former position. A relative device can be used to specify an arbitrarily large change in position: A user can move a mouse along the desk top, lift it up and place it back at its initial starting position, and move it again. A data tablet can be programmed to behave as a relative device: The first (x, y) coordinate position read after the pen goes from "far" to "near" state (i.e., close to the tablet) is subtracted from all subsequently read coordinates to yield only the change in x and y, which is added to the previous (x, y) position. This process is continued until the pen again goes to "far" state.

Relative devices cannot be used readily for digitizing drawings, whereas absolute devices can be. The advantage of a relative device is that the application program can reposition the cursor anywhere on the screen.

With a *direct* device—such as a light pen or touch screen—the user points directly at the screen with a finger or surrogate finger; with an *indirect* device—such as a tablet, mouse, or joystick—the user moves a cursor on the screen using a device not on the screen. New forms of eye–hand coordination must be learned for the latter; the proliferation of computer games in homes and arcades, however, is creating an environment in which many casual computer users have already learned these skills. However, direct pointing can cause arm fatigue, especially among casual users.

Samsung/Google, EX1027, p. 366

A *continuous* device is one in which a smooth hand motion can create a smooth cursor motion. Tablets, joysticks, and mice are all continuous devices, whereas cursor-control keys are *discrete* devices. Continuous devices typically allow more natural, easier, and faster cursor movement than do discrete devices. Most continuous devices also permit easier movement in arbitrary directions than do cursor control keys.

Speed of cursor positioning with a continuous device is affected by the *control-to-display ratio*, commonly called the C/D ratio [CHAP72]; it is the ratio between hand movement (the control) and cursor movement (the display). A large ratio is good for accurate positioning, but makes rapid movements tedious; a small ratio is good for speed but not for accuracy. Fortunately, for a relative positioning device, the ratio need not be constant, but can be changed adaptively as a function of control-movement speed. Rapid movements indicate the user is making a gross hand movement, so a small ratio is used; as the speed decreases, the C/D ratio is increased. This variation of C/D ratio can be set up so that a user can use a mouse to position a cursor accurately across a 15-inch screen without repositioning her wrist! For indirect discrete devices (cursor-control keys), there is a similar technique: the distance the cursor is moved per unit time is increased as a function of the time the key has been held down.

Precise positioning is difficult with direct devices, if the arm is unsupported and extended toward the screen. Try writing your name on a blackboard in this pose, and compare the result to your normal signature. This problem can be mitigated if the screen is angled close to horizontal. Indirect devices, on the other hand, allow the heel of the hand to rest on a support, so that the fine motor control of the fingers can be used more effectively. Not all continuous indirect devices are equally satisfactory for drawing, however. Try writing your name with a joystick, a mouse, and a tablet pen stylus. Using the stylus is fastest, and the result is most pleasing.

Other interesting positioning devices include the Versatron *footmouse* [VERS84], which remains static on the floor: The user places the ball of his foot on the device, keeping his heel on the floor, and controls the footmouse with left–right and forward–backward movements. The experimental *mole* is a pivoted foot rest with integrated switches [PEAR86, PEAR88] that, like the footmouse, leaves the hands free. The Personics *headmouse* [PERS85] uses a head-mounted set of three microphones to measure the distance to a sound source, translating small rotational movements of the head into cursor movements. *Eye trackers* can determine where the eye is pointing and hence can cause a cursor to move or the object pointed at to be selected [BOLT80; BOLT84; WARE87]. These devices are often less accurate and considerably more expensive than are the more traditional devices, and thus would normally be considered for only hands-free applications. The 3D positioning devices discussed in Section 8.1.6 can also be used for 2D positioning.

8.1.2 Keyboard Devices

The well-known QWERTY keyboard has been with us for many years. It is ironic that this keyboard was originally designed to *slow down* typists, so that the typewriter hammers would not be so likely to jam. Studies have shown that the newer Dvořák keyboard [DVOR43], which places vowels and other high-frequency characters under the home

Samsung/Google, EX1027, p. 367

positions of the fingers, is somewhat faster than is the QWERTY design [GREE87]. It has not been widely accepted. Alphabetically organized keyboards are sometimes used when many of the users are nontypists. But more and more people are being exposed to QWERTY keyboards, and several experiments have shown no advantage of alphabetic over QWERTY keyboards [HIRS70; MICH71].

The *chord keyboard* has five keys similar to piano keys, and is operated with one hand, by pressing one or more keys simultaneously to "play a chord." With five keys, 31 different chords can be played. Learning to use a chord keyboard (and other similar stenographer-style keyboards) takes longer than learning the QWERTY keyboard, but skilled users can type quite rapidly, leaving the second hand free for other tasks. This increased training time means, however, that such keyboards are not suitable substitutes for general use of the standard alphanumeric keyboard.

Other keyboard-oriented considerations, involving not hardware but software design, are arranging for a user to enter frequently used punctuation or correction characters without needing simultaneously to press the control or shift keys, and assigning dangerous actions (such as delete) to keys that are distant from other frequently used keys.

8.1.3 Valuator Devices

Some valuators are *bounded*, like the volume control on a radio—the dial can be turned only so far before a stop is reached that prevents further turning. A bounded valuator inputs an absolute quantity. A continuous-turn potentiometer, on the other hand, can be turned an *unbounded* number of times in either direction. Given an initial value, the unbounded potentiometer can be used to return absolute values; otherwise, the returned values are treated as relative values. The provision of some sort of echo enables the user to determine what relative or absolute value is currently being specified. The issue of C/D ratio, discussed in the context of positioning devices, also arises in the use of slide and rotary potentiometers to input values.

8.1.4 Choice Devices

Function keys are a common choice device. Their placement affects their usability: keys mounted on the CRT bezel are harder to use than are keys mounted in the keyboard or in a nearby separate unit. A foot switch can be used in applications in which the user's hands are engaged yet a single switch closure must be frequently made. For example, used with a headmouse (described in Section 8.1.1), a foot switch could easily provide functionality equivalent to a single-button mouse.

8.1.5 Other Devices

Here we discuss some of the less common, and in some cases experimental, 2D interaction devices. Voice recognizers, which are useful because they free the user's hands for other uses, apply a pattern-recognition approach to the waveforms created when we speak a word. The waveform is typically separated into a number of different frequency bands, and the variation over time of the magnitude of the waveform in each band forms the basis for the pattern matching. However, mistakes can occur in the pattern matching, so it is especially

Samsung/Google, EX1027, p. 368

important that an application using a recognizer provide convenient correction capabilities.

Voice recognizers differ in whether or not they must be trained to recognize the waveforms of a particular speaker, and whether they can recognize connected speech as opposed to single words or phrases. Speaker-independent recognizers have very limited vocabularies—typically, they include only the digits and 50 to 100 words. Some discrete-word recognizers can recognize vocabularies of up to a few thousand different words after appropriate training. But if the user has a cold, the recognizer must be retrained. The user of a recognizer must pause for a fraction of a second after each word to cue the system that a word end has occurred; the pause is typically 100 to 200 milliseconds, and can be longer if the set of possible words is large. The more difficult task of recognizing connected speech from a limited vocabulary can also be performed by commercial hardware, but at a higher cost. The larger the vocabulary, however, the more artificial-intelligence techniques are needed to use the context and meaning of a sequence of sentences to remove ambiguity. A few systems with vocabularies of 20,000 or more words can recognize sentences such as "Write Mrs. Wright a letter right now!"

Voice synthesizers create waveforms that approximate, with varying degrees of realism, spoken words [KAPL85]. The simplest synthesizers use *phonemes*, the basic sound units that form words. This approach creates an artificial-sounding, inflection-free voice. More sophisticated phoneme-based systems add inflections. Other systems actually play back digitized speech patterns. They sound realistic, but require thousands of bytes of memory to store the digitized speech.

Now that several personal computers, including the Macintosh and NeXT, have standard sound synthesizers that can create both voice and music, speech feedback from computers is becoming quite common. Speech is best used to augment rather than to replace visual feedback, and is most effective when used sparingly. For instance, a training application could show a student a graphic animation of some process, along with a voice narration describing what is being seen. See [SIMP85; SIMP87] for an extensive review of speech recognition and generation, including additional guidelines for the effective application of these functions in user–computer interfaces.

Sound generators can be used to generate musical tones and other effects, which can call attention to specific situations, especially if the user is unlikely to be looking at the display. For instance, "printer out of paper" or "memory nearly full" alarms might be signaled by two different tones, in addition to messages on the screen. An attempt to reorient a line that has been constrained to be parallel to another line might cause a warning beep.

The data tablet has been extended in several ways. Many years ago, Herot and Negroponte used an experimental pressure-sensitive stylus [HERO76]: High pressure and a slow drawing speed implied that the user was drawing a line with deliberation, in which case the line was recorded exactly as drawn; low pressure and fast speed implied that the line was being drawn quickly, in which case a straight line connecting the endpoints was recorded. A more recent commercially available tablet [GTCO82] senses not only stylus pressure but orientation as well. The resulting 5 degrees of freedom reported by the tablet can be used in various creative ways. For example, Bleser, Sibert, and McGee implemented the GWPaint system to simulate various artist's tools, such as an italic pen, that are sensitive to pressure and orientation [BLES88a]. Figure 8.1 shows the artistic creativity thus afforded.

Samsung/Google, EX1027, p. 369



Fig. 8.1 *Numeral 2*, a drawing in the spirit of Jasper Johns, by Teresa Bleser. Drawn with the GWPaint program using a GTCO pressure- and tilt-sensitive tablet. (Courtesy of T. Bleser, George Washington University.)

Green [GREE85] applied optics principles to develop a tabletlike device that gives artists even more freedom than the pressure- and tilt-sensing tablet. The user paints on the tablet with brushes, hands, or anything else that is convenient. A television camera positioned below the tablet records the shape and motion of the brush wherever it contacts the tablet, and the resulting video signal is used to load the refresh buffer of a raster display. Resolution of 500 or 1000 units is achievable, depending on the television camera used.

An experimental touch tablet, developed by Buxton and colleagues, can sense multiple finger positions simultaneously, and can also sense the area covered at each point of contact [LEE85a]. The device is essentially a type of touch panel, but is used as a tablet on the work surface, not as a touch panel mounted over the screen. The device can be used in a rich variety of ways [BUXT85]. Different finger pressures correlate with the area covered at a point of contact, and are used to signal user commands: a light pressure causes a cursor to appear and to track finger movement; increased pressure is used, like a button-push on a mouse or puck, to begin feedback such as dragging of an object; decreased pressure causes the dragging to stop.

Samsung/Google, EX1027, p. 370

Another way to obtain more than just position information is to suspend a touch panel in front of a display using a few metal support strips with strain gauges [HERO78, MINS84]. Pressure applied to the touch panel translates into strain measured by the gauges. It is possible also to measure the direction of push and torque as well, by orienting the metal strips in appropriate directions. The measurements can be used to cause displayed objects to be rotated, scaled, and so on.

The decreasing costs of input devices and the increasing availability of computer power are likely to lead to the continuing introduction of novel interaction devices. Douglas Engelbart invented the mouse in the 1960s, and nearly 20 years passed before it became popular [PERR89]. What the next mouse will be is not yet clear, but we hope that it will have a much shorter gestation period.

8.1.6 3D Interaction Devices

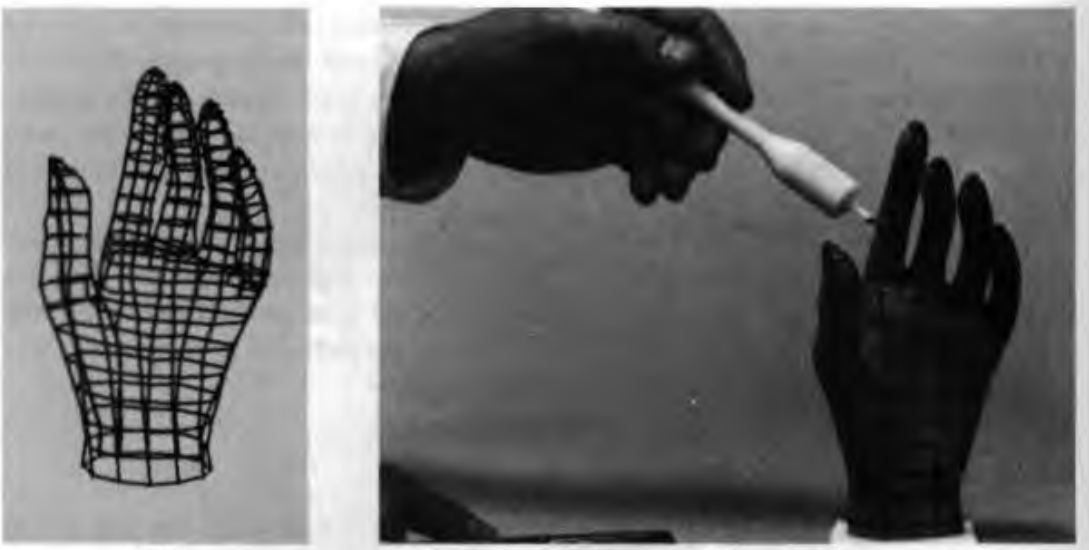
Some of the 2D interaction devices are readily extended to 3D. Joysticks can have a shaft that twists for a third dimension (see Fig. 4.38). Trackballs can be made to sense rotation about the vertical axis in addition to that about the two horizontal axes. In both cases, however, there is no direct relationship between hand movements with the device and the corresponding movement in 3-space.

The Spaceball (see Color Plate I.14) is a rigid sphere containing strain gauges. The user pushes or pulls the sphere in any direction, providing 3D translation and orientation. In this case, at least the directions of movement correspond to the user's attempts at moving the rigid sphere, although the hand does not actually move.

A number of devices, on the other hand, can record 3D hand movements. The experimental *Noll Box*, developed by Michael Noll, permits movement of a knob in a 12-inch cube volume, sensed by slider mechanisms linked to potentiometers. The Polhemus 3SPACE three-dimensional position and orientation sensor uses electromagnetic coupling between three transmitter antennas and three receiver antennas. The transmitter antenna coils, which are at right angles to one another to form a Cartesian coordinate system, are pulsed in turn. The receiver has three similarly arranged receiver antennas; each time a transmitter coil is pulsed, a current is induced in each of the receiver coils. The strength of the current depends both on the distance between the receiver and transmitter and on the relative orientation of the transmitter and receiver coils. The combination of the nine current values induced by the three successive pulses is used to calculate the 3D position and orientation of the receiver. Figure 8.2 shows this device in use for one of its common purposes: digitizing a 3D object.

The DataGlove records hand position and orientation as well as finger movements. As shown in Fig. 8.3, it is a glove covered with small, lightweight sensors. Each sensor is a short length of fiberoptic cable, with a light-emitting diode (LED) at one end and a phototransistor at the other end. The surface of the cable is roughened in the area where it is to be sensitive to bending. When the cable is flexed, some of the LED's light is lost, so less light is received by the phototransistor. In addition, a Polhemus position and orientation sensor records hand movements. Wearing the DataGlove, a user can grasp objects, move and rotate them, and then release them, thus providing very natural interaction in 3D [ZIMM87]. Color Plate I.15 illustrates this concept.

Samsung/Google, EX1027, p. 371



(a) (b)

Fig. 8.2 (a) A wireframe display of the result. (b) The Polhemus 3D position sensor being used to digitize a 3D object. (3Space digitizer courtesy of Polhemus, Inc., Colchester, VT.)

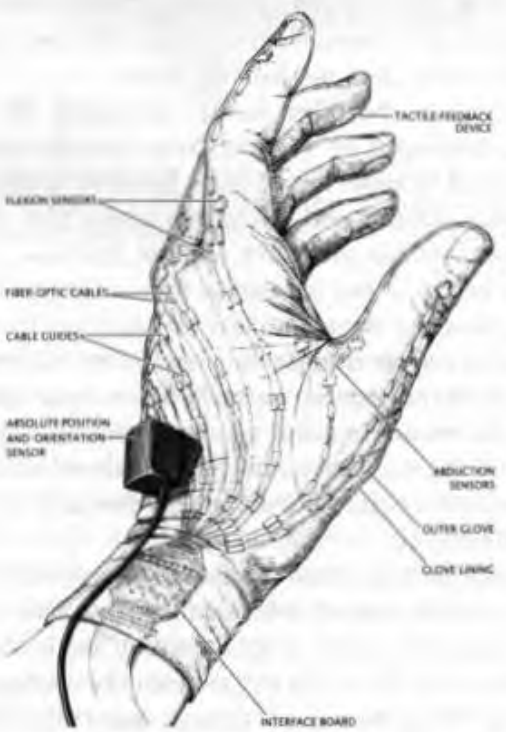


Fig. 8.3 The VPL DataGlove, showing the fiberoptic calbes that are used to sense finger movements, and the Polhemus position and orientation sensor. (From J. Foley, Interfaces for Advanced Computing, Copyright © 1987 by *SCIENTIFIC AMERICAN, Inc.* All rights reserved.)

Samsung/Google, EX1027, p. 372

Considerable effort has been directed toward creating what are often called *artificial realites* or *virtual realities*; these are completely computer-generated environments with realistic appearance, behavior, and interaction techniques [FOLE87]. In one version, the user wears a head-mounted stereo display to show proper left- and right-eye views, a Polhemus sensor on the head allows changes in head position and orientation to cause changes to the stereo display, a DataGlove permits 3D interaction, and a microphone is used for issuing voice commands. Color Plate I.16 shows this combination of equipment.

Several other technologies can be used to record 3D positions. The sonic-tablet technology discussed in Section 4.6.1 can be extended to 3D to create a *sonic pen*. In one approach, three orthogonal strip microphones are used. The hand-held pen sparks 20 to 40 times per second, and the time for the sound to arrive at each of the three microphones determines the radius of the three cylinders on which the pen is located. The location is thus computed as the intersection of three cylinders. A similar approach uses three or four standard microphones; here, the pen location is computed as the intersection of spheres with centers at the microphones and radii determined by the time the sound takes to arrive at each microphone.

All these systems work in relatively small volumes—8 to 27 cubic feet. Optical sensors can give even greater freedom of movement [BURT74; FUCH77a]. LEDs are mounted on the user (either at a single point, such as the fingertip, or all over the body, to measure body movements). Light sensors are mounted high in the corners of a small, semidarkened room in which the user works, and each LED is intensified in turn. The sensors can determine the plane in which the LED lies, and the location of the LED is thus at the intersection of three planes. (A fourth sensor is normally used, in case one of the sensors cannot see the LED.) Small reflectors on the fingertips and other points of interest can replace the LEDs; sensors pick up reflected light rather than the LED's emitted light.

Krueger [KRUE83] has developed a sensor for recording hand and finger movements in 2D. A television camera records hand movements; image-processing techniques of contrast-enhancement and edge detection are used to find the outline of the hand and fingers. Different finger positions can be interpreted as commands, and the user can grasp and manipulate objects, as in Color Plate I.17. This technique could be extended to 3D through use of multiple cameras.

8.1.7 Device-Level Human Factors

Not all interaction devices of the same type are equivalent from a human-factors point of view (see [BUXT86] for an elaboration of this theme). For instance, mice differ in important ways. First, the physical shapes are different, ranging from a hemisphere to an elongated, low-profile rectangle. Buttons are positioned differently. Buttons on the side or front of a mouse may cause the mouse to move a bit when the buttons are pressed; buttons on the top of a mouse do not have this effect. The mouse is moved through small distances by wrist and finger movements, with the fingers grasping the mouse toward its front. Yet the part of the mouse whose position is sensed is often toward the rear, where fine control is least possible. In fact, a small leftward movement of the mouse under the fingertips can include a bit of rotation, so that the rear of the mouse, where the position sensors are, actually moves a bit to the right!

Samsung/Google, EX1027, p. 373

There is great variation among keyboards in design parameters, such as keycap shape, distance between keys, pressure needed to press a key, travel distance for key depression, key bounce, auditory feedback, the feeling of contact when the key is fully depressed, and the placement and size of important keys such as "return" or "enter." Improper choice of parameters can decrease productivity and increase error rates. Making the "return" key too small invites errors, as does placing a hardware "reset" key close to other keys. These and other design parameters are discussed in [KLEM71; GREE87], and have been the subject of recent international standardization efforts.

The tip of a short joystick shaft moves through a short distance, forcing use of a small C/D ratio; if we try to compensate by using a longer joystick shaft, the user cannot rest the heel of her hand on the work surface and thus does not have a steady platform from which to make fine adjustments. Accuracy and speed therefore suffer.

The implication of these device differences is that it is not enough for a user interface designer to specify a particular device class; specific device characteristics must be defined. Unfortunately, not every user interface designer has the luxury of selecting devices; often, the choice has already been made. Then the designer can only hope that the devices are well designed, and attempt to compensate in software for any hardware deficiencies.

8.2 BASIC INTERACTION TASKS

With a basic interaction task, the user of an interactive system enters a unit of information that is meaningful in the context of the application. How large or small is such a unit? For instance, does moving a positioning device a small distance enter a unit of information? Yes, if the new position is put to some application purpose, such as repositioning an object or specifying the endpoint of a line. No, if the repositioning is just one of a sequence of repositionings as the user moves the cursor to place it on top of a menu item: here, it is the menu choice that is the unit of information.

Basic interaction tasks (BITs) are indivisible; that is, if they were decomposed into smaller units of information, the smaller units would not in themselves be meaningful to the application. BITs are discussed in this section. In the next section, we treat composite interaction tasks (CITs), which are aggregates of the basic interaction tasks described here. If one thinks of BITs as atoms, then CITs are molecules.

A complete set of BITs for interactive graphics is positioning, selecting, entering text, and entering numeric quantities. Each BIT is described in this section, and some of the many interaction techniques for each are discussed. However, there are far too many interaction techniques for us to give an exhaustive list, and we cannot anticipate the development of new techniques. Where possible, the pros and cons of each technique are discussed; remember that a specific interaction technique may be good in some situations and poor in others.

8.2.1 The Position Interaction Task

The positioning task involves specifying an (x, y) or (x, y, z) position to the application program. The customary interaction techniques for carrying out this task involve either moving a screen cursor to the desired location and then pushing a button, or typing the

Samsung/Google, EX1027, p. 374

desired position's coordinates on either a real or a simulated keyboard. The positioning device can be direct or indirect, continuous or discrete, absolute or relative. In addition, cursor-movement commands can be typed explicitly on a keyboard, as Up, Left, and so on, or the same commands can be spoken to a voice-recognition unit. Furthermore, techniques can be used together—a mouse controlling a cursor can be used for approximate positioning, and arrow keys can be used to move the cursor a single screen unit at a time for precise postioning.

A number of general issues transcend any one interaction technique. We first discuss the general issues; we introduce specific positioning techniques as illustrations.

Coordinate systems. An important issue in positioning is the coordinate system in which feedback is provided. If a locator device is moved to the right to drag an object, in which direction should the object move? There are at least three possibilities: the object could move along the increasing x direction in the screen-coordinate system, along the increasing x direction in world coordinates, or along the increasing x direction in the object's own coordinate system.

The first alternative, increasing screen-coordinate x direction, is the correct choice. For the latter two options, consider that the increasing x direction need not in general be along the screen coordinates' x axis. For instance, if the viewing transformation includes a 180° rotation, then the world coordinates' x axis goes in the opposite direction to the screen coordinates' x axis, so that the right-going movement of the locator would cause a left-going movement of the object. Try positioning with this type of feedback by turning your mouse 180°! Such a system would be a gross violation of the human-factors principle of stimulus-response compatibility (S-R compatibility), which states that system responses to user actions must be in the same direction or same orientation, and that the magnitude of the responses should be proportional to the actions. Similar problems can occur if a data tablet is rotated with respect to the screen.

Resolution. The resolution required in a positioning task may vary from one part in a few hundred to one part in millions. Clearly, keyboard typein of an (x, y) pair can provide unlimited resolution: The typed digit strings can be as long as necessary. What resolution can cursor-movement techniques achieve? The resolution of tablets, mice, and so on is typically as least as great as the 500 to 2000 resolvable units of the display device. By using the window-to-viewport transformation to zoom in on part of the world, it is possible to arrange for one unit of screen resolution to correspond to an arbitrarily small unit of world-coordinate resolution.

Touch panels present other interesting resolution issues. Some panels are accurate to 1000 units. But the user's finger is about $\frac{1}{2}$-inch wide, so how can this accuracy be achieved? Using the first position the finger touches as the final position does not work. The user must be able to drag a cursor around on the screen by moving or rolling his finger while it is in contact with the touch panel. Because the finger obscures the exact position being indicated, the cursor arms can be made longer than normal, or the cursor can be offset from the actual point of contact. In an experiment, dragging an offset cursor was found to be more accurate, albeit slower, than was using the first point contacted [POTT88]. In general, the touch panel is not recommended for frequent high-resolution positioning tasks.

Samsung/Google, EX1027, p. 375

Grids. An important visual aid in many positioning tasks is a grid superimposed (perhaps at low intensity) on the work area, to help in aligning positions or objects. It can also be useful to force endpoints of primitives to fall on the grid, as though each grid point were surrounded by a gravity field. Gridding helps users to generate drawings with a neat appearance. To enforce gridding, the application program simply rounds locator coordinates to the nearest grid point (in some cases, only if the point is already close to a grid point). Gridding is usually applied in world coordinates. Although grids often are regular and span the entire display, irregular grids, different grids in different areas, as well as rotated grids, are all useful in creating figures and illustrations [BIER86a; FEIN82a].

Feedback. There are two types of positioning tasks, spatial and linguistic. In a *spatial* positioning task, the user knows where the intended position is, in spatial relation to nearby elements, as in drawing a line between two rectangles or centering an object between two others. In a *linguistic* positioning task, the user knows the numeric values of the (x, y) coordinates of the position. In the former case, the user wants feedback showing the actual position on the screen; in the latter case, the coordinates of the position are needed. If the wrong form of feedback is provided, the user must mentally convert from one form to the other. Both forms of feedback can be provided by displaying both the cursor and its numeric coordinates, as in Fig 8.4.

Direction preference. Some positioning devices impede movement in arbitrary directions; for example, certain joysticks and joyswitches give more resistance to movements off the principal axes than they do to those on the axes. This is useful only if the positioning task itself is generally constrained to horizontal and vertical movements.

Learning time. Learning the eye–hand coordination for indirect methods is essentially the same process as learning to steer a car. Learning time is a common concern but turns out to be a minor issue. Card and colleagues [CARD78] studied the mouse and joystick. They found that, although practice improved both error rates and speed, even the novices' performance was quite good. For instance, selection time with a mouse (move cursor to target, press button) decreased with extensive practice from 2.2 to 1.7 seconds. It is true, however, that some users find the indirect coordination very difficult, until they are explicitly taught.

One specific type of postioning task is *continuous positioning*, in which a sequence of positions is used to define a curve. The path taken by the locator is approximated by a connected series of very short lines, as shown in Fig. 8.5. So that the appearance of



Fig. 8.4 Numeric feedback regarding size of an object being constructed. The height and width are changed as the cursor (+) is moved, so the user can adjust the object to the desired size.

Samsung/Google, EX1027, p. 376

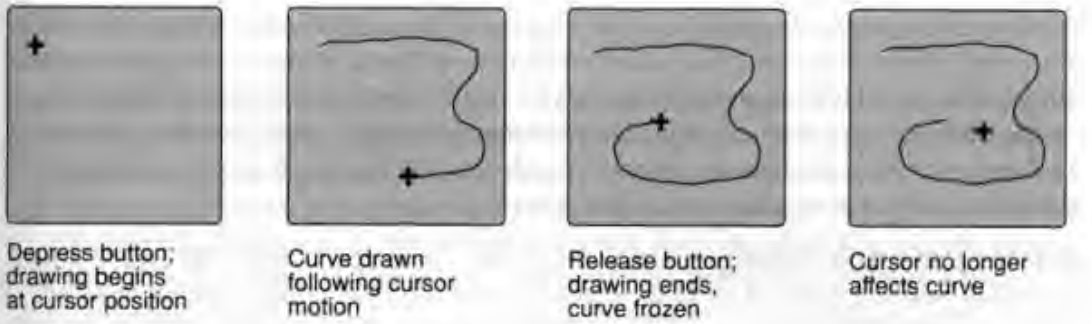


Fig. 8.5 Continuous sketching.

smoothness is maintained, more lines may be used where the radius of curvature is small, or individual dots may be displayed on the cursor's path, or a higher-order curve can be fitted through the points (see Chapter 11).

Precise continuous positioning is easier with a stylus than with a mouse, because the stylus can be controlled precisely with finger muscles, whereas the mouse is controlled primarily with wrist muscles. Digitizing of drawings is difficult with a mouse for the same reason; in addition, the mouse lacks both an absolute frame of reference and a cross-hair. On the other hand, a mouse requires only a small table area and is less expensive than a tablet.

8.2.2 The Select Interaction Task—Variable-Sized Set of Choices

The selection task is that of choosing an element from a *choice set*. Typical choice sets are commands, attribute values, object classes, and object instances. For example, the line-style menu in a typical paint program is a set of attribute values, and the object-type (line, circle, rectangle, text, etc.) menu in such programs is a set of object classes. Some interaction techniques can be used to select from any of these four types of choice sets; others are less general. For example, pointing at a visual representation of a set element can serve to select it, no matter what the set type. On the other hand, although function keys often work quite well for selecting from a command, object class, or attribute set, it is difficult to assign a separate function key to each object instance in a drawing, since the size of the choice set is variable, often is large (larger than the number of available function keys), and changes quite rapidly as the user creates and deletes objects.

We use the terms *(relatively) fixed-sized choice set* and *varying-sized choice set*. The first term characterizes command, attribute, and object-class choice sets; the second, object-instance choice sets. The "relatively" modifier recognizes that any of these sets can change as new commands, attributes, or object classes (such as symbols in a drafting system) are defined. But the set size does not change frequently, and usually does not change much. Varying-sized choice sets, on the other hand, can become quite large, and can change frequently.

In this section, we discuss techniques that are particularly well suited to potentially large varying-sized choice sets; these include naming and pointing. In the following section, we discuss selection techniques particularly well suited to (relatively) fixed-sized choice sets. These sets tend to be small, except for the large (but relatively fixed-sized) command

Samsung/Google, EX1027, p. 377

sets found in complex applications. The techniques discussed include typing or speaking the name, abbreviation, or other code that represents the set element; pressing a function key associated with the set element (this can be seen as identical to typing a single character on the keyboard); pointing at a visual representation (textual or graphical) of the set element in a menu; cycling through the set until the desired element is displayed; and making a distinctive motion with a continuous positioning device.

Selecting objects by naming. The user can type the choice's name. The idea is simple, but what if the user does not know the object's name, as could easily happen if hundreds of objects are being displayed, or if the user has no reason to know names? Nevertheless, this technique is useful in several situations. First, if the user is likely to know the names of various objects, as a fleet commander would know the names of the fleet's ships, then referring to them by name is reasonable, and can be faster than pointing, especially if the user might need to scroll through the display to bring the desired object into view. Second, if the display is so cluttered that picking by pointing is difficult *and* if zooming is not feasible (perhaps because the graphics hardware does not support zooming and software zoom is too slow), then naming may be a choice of last resort. If clutter is a problem, then a command to turn object names on and off would be useful.

Typing allows us to make multiple selections through wild-card or don't-care characters, if the choice set elements are named in a meaningful way. Selection by naming is most appropriate for experienced, regular users, rather than for casual, infrequent users.

If naming by typing is necessary, a useful form of feedback is to display, immediately after each keystroke, the list (or partial list, if the full list is too long) of names in the selection set matching the sequence of characters typed so far. This can help the user to remember just how the name is spelled, if he has recalled the first few characters. As soon as an unambiguous match has been typed, the correct name can be automatically highlighted on the list. Alternatively, the name can be automatically completed as soon as an unambiguous match has been typed. This technique, called *autocompletion*, is sometimes disconcerting to new users, so caution is advisable. A separate strategy for name typein is spelling correction (sometimes called *Do What I Mean*, or DWIM). If the typed name does not match one known to the system, other names that are close to the typed name can be presented to the user as alternatives. Determining closeness can be as simple as searching for single-character errors, or can include multiple-character and missing-character errors.

With a voice recognizer, the user can speak, rather than type, a name, abbreviation, or code. Voice input is a simple way to distinguish commands from data: Commands are entered by voice, the data are entered by keyboard or other means. In a keyboard environment, this eliminates the need for special characters or modes to distinguish data and commands.

Selecting objects by pointing. Any of the pointing techniques mentioned in the introduction to Section 8.2 can be used to select an object, by first pointing and then indicating (typically via a button-push) that the desired object is being pointed at. But what if the object has multiple levels of hierarchy, as did the robot of Chapter 7? If the cursor is over the robot's hand, it is not clear whether the user is pointing at the hand, the arm, or the entire robot. Commands like Select_robot and Select_arm can be used to specify the level of hierarchy. On the other hand, if the level at which the user works changes infrequently, the

Samsung/Google, EX1027, p. 378

user will be able to work faster with a separate command, such as Set_selection_level, used to change the level of hierarchy.

A different approach is needed if the number of hierarchical levels is unknown to the system designer and is potentially large (as in a drafting system, where symbols are made up of graphics primitives and other symbols). At least two user commands are required: Up_hierarchy and Down_hierarchy. When the user selects something, the system highlights the lowest-level object seen. If this is what he desired, the user can proceed. If not, the user issues the first command: Up_hierarchy. The entire first-level object of which the detected object is a part is highlighted. If this is not what the user wants, he travels up again and still more of the picture is highlighted. If he travels too far up the hierarchy, he reverses direction with the Down_hierarchy command. In addition, a Return_to_lowest_level command can be useful in deep hierarchies, as can a hierarchy diagram in another window, showing where in the hierarchy the current selection is located. The state diagram of Fig. 8.6

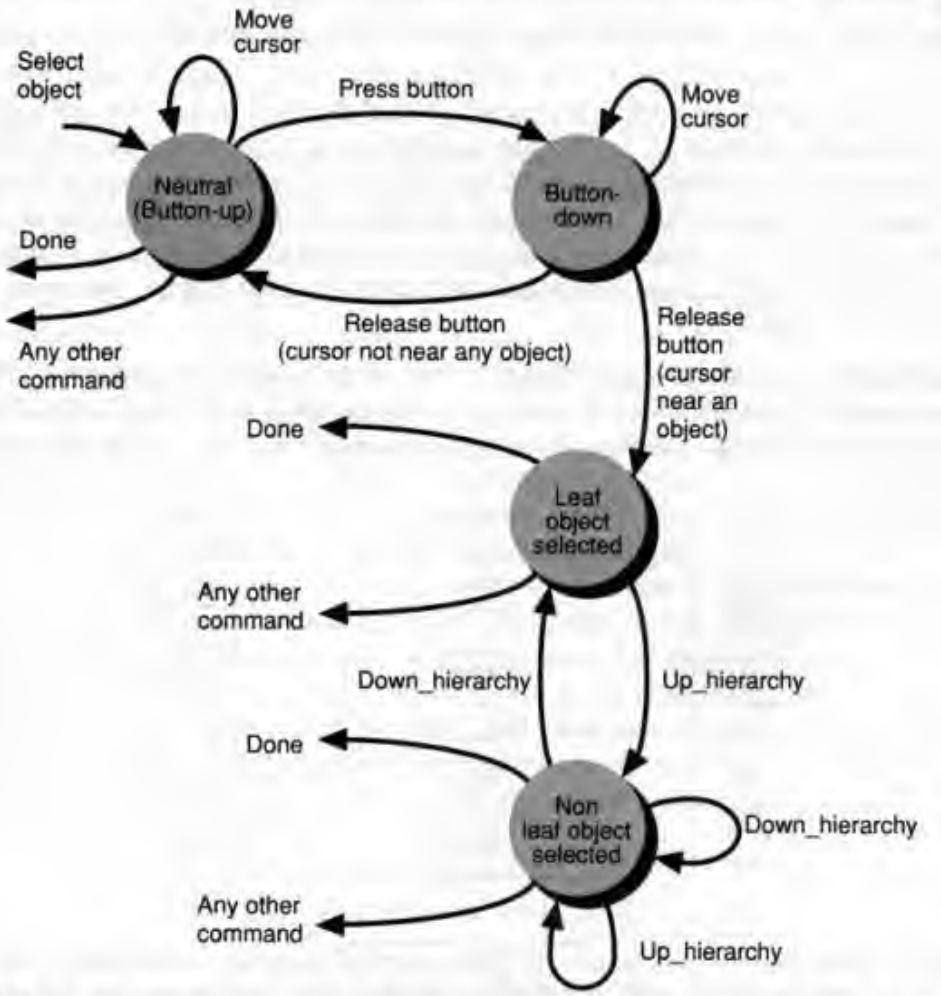


Fig. 8.6 State diagram for an object-selection technique for an arbitrary number of hierarchy levels. Up and Down are commands for moving up and down the hierarchy. In the state "Leaf object selected," the Down_hierarchy command is not available. The user selects an object by pointing at it with a cursor, and pressing and then releasing a button.

Samsung/Google, EX1027, p. 379

shows one approach to hierarchical selection. Alternatively, a single command, say Move_up_hierarchy, can skip back to the originally selected leaf node after the root node is reached.

Some text editors use a character–word–sentence–paragraph hierarchy. In the Xerox Star text editor, for instance, the user selects a character by positioning the screen cursor on the character and clicking the Select button on the mouse. To choose the word rather than the character, the user clicks twice in rapid succession. Further moves up the hierarchy are accomplished by additional rapid clicks.

8.2.3 The Select Interaction Task—Relatively Fixed-Sized Choice Set

Menu selection is one of the richest techniques for selecting from a relatively fixed-sized choice set. Here we discuss several key factors in menu design.

Menu order. Menu elements can be organized in many different orders, including alphabetical, logically grouped by functional purpose, most frequently used first, most important first, largest first, or most recently created/modified first. These orders can be combined in various ways. A functionally grouped menu may be ordered alphabetically within group, and the functional groups themselves ordered by frequency of use. Figure 8.7 illustrates several such possible organizations. Consistency of organization from one menu to another is useful, so a common strategy across all menus of an application is important. Several researchers have found functional order to be the most helpful, and many menu structures reflect this result.

Single-level versus hierarchical design. One of the most fundamental menu design decisions arises if the choice set is too large to display all at once. Such a menu can be subdivided into a logically structured hierarchy or presented as a linear sequence of choices

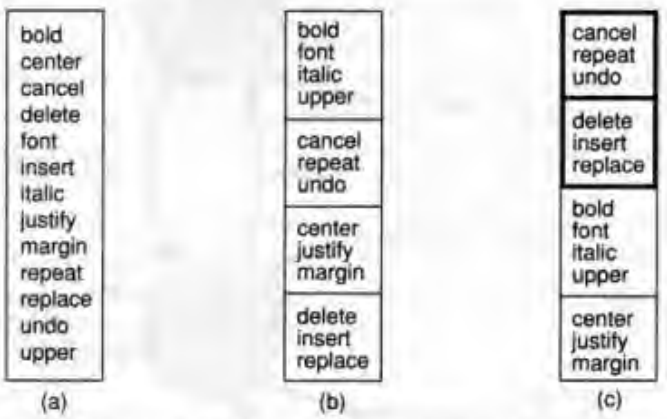


Fig. 8.7 Three menu organizations. (a) Menu using an alphabetical sequence. (b) Menu using functional grouping, with alphabetical within-group order as well as alphabetical-between-group order. (c) Menu with commands common to several different application programs placed at the top for consistency with the other application's menus; these commands have heavier borders. Menu items are some of those used in Card's menu-order experiment [CARD82].

Samsung/Google, EX1027, p. 380

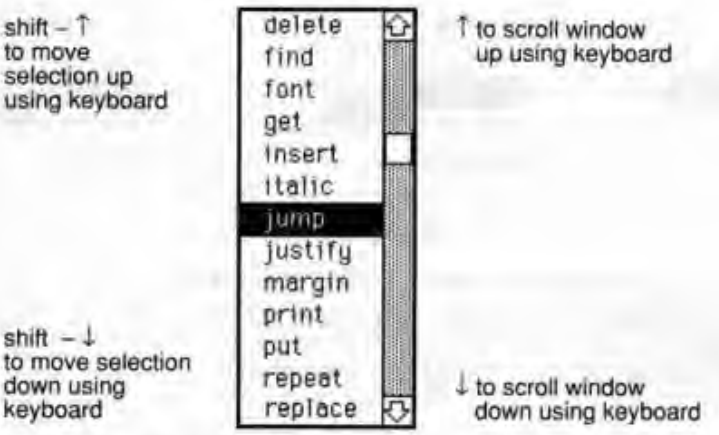


Fig. 8.8 A menu within a scrolling window. The user controls scrolling by selecting the up and down arrows or by dragging the square in the scroll bar.

to be paged or scrolled through. A scroll bar of the type used in many window managers allows all the relevant scrolling and paging commands to be presented in a concise way. A fast keyboard-oriented alternative to pointing at the scrolling commands can also be provided; for instance, the arrow keys can be used to scroll the window, and the shift key can be combined with the arrow keys to move the selection within the visible window, as shown in Fig. 8.8. In the limit, the size of the window can be reduced to a single menu item, yielding a "slot-machine" menu of the type shown in Fig. 8.9.

With a hierarchical menu, the user first selects from the choice set at the top of the hierarchy, which causes a second choice set to be available. The process is repeated until a leaf node (i.e., an element of the choice set itself) of the hierarchy tree is selected. As with hierarchical object selection, navigation mechanisms need to be provided so that the user can go back up the hierarchy if an incorrect subtree was selected. Visual feedback to give the user some sense of place within the hierarchy is also needed.

Menu hierarchies can be presented in several ways. Of course, successive levels of the hierarchy can replace one another on the display as further choices are made, but this does not give the user much sense of position within the hierarchy. The *cascading hierarchy*, as depicted in Fig. 8.10, is more attractive. Enough of each menu must be revealed that the complete highlighted selection path is visible, and some means must be used to indicate whether a menu item is a leaf node or is the name of a lower-level menu (in the figure, the right-pointing arrow fills this role). Another arrangement is to show just the name of each

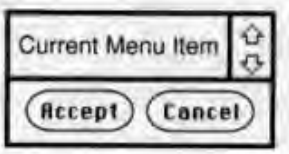


Fig. 8.9 A small menu-selection window. Only one menu item appears at a time. The scroll arrows are used to change the current menu item, which is selected when the Accept button is chosen.

Samsung/Google, EX1027, p. 381

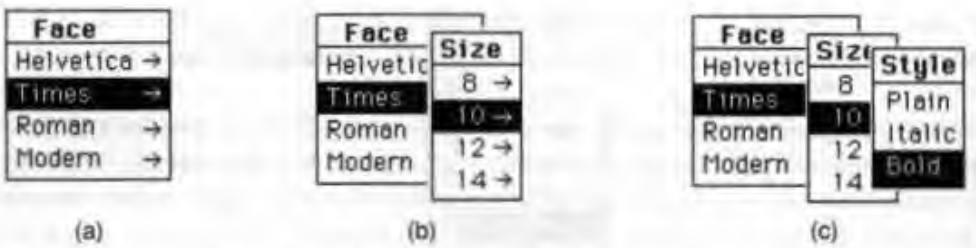


Fig. 8.10 A pop-up hierarchical menu. (a) The first menu appears where the cursor is, in response to a button-down action. The cursor can be moved up and down to select the desired typeface. (b) The cursor is then moved to the right to bring up the second menu. (c) The process is repeated for the third menu.

selection made thus far in traversing down the hierarchy, plus all the selections available at the current level.

A *panel hierarchy* is another way to depict a hierarchy, as shown in Fig. 8.11; it takes up somewhat more room than the cascading hierarchy. If the hierarchy is not too large, an explicit tree showing the entire hierarchy can also be displayed.

When we design a hierarchical menu, the issue of depth versus breadth is always present. Snowberry et al. [SNOW83] found experimentally that selection time and accuracy improve when broader menus with fewer levels of selection are used. Similar results are reported by Landauer and Nachbar [LAND85] and by other researchers. However, these



Fig. 8.11 A hierarchical-selection menu. The leftmost column represents the top level; the children of the selected item in this column are shown in the next column; and so on. If there is no selected item, then the columns to the right are blank. (Courtesy of NeXT, Inc. © 1989 NeXT, Inc.)

Samsung/Google, EX1027, p. 382

results do not necessarily generalize to menu hierarchies that lack a natural, understandable structure.

Hierarchical menu selection almost demands an accompanying keyboard or function-key accelerator technique to speed up selection for more experienced (so-called "power") users. This is easy if each node of the tree has a unique name, so that the user can enter the name directly, and the menu system provides a backup if the user's memory fails. If the names are unique only within each level of the hierarchy, the power user must type the complete path name to the desired leaf node.

Menu placement. Menus can be shown on the display screen or on a second, auxiliary screen (Fig. 8.12); they can also be printed on a tablet or on function-key labels. Onscreen menus can be static and permanently visible, or can appear dynamically on request (tear-off, appearing, pop-up, pull-down, and pull-out menus).

A static menu printed on a tablet, as shown in Color Plate I.18, can easily be used in fixed-application systems. Use of a tablet or an auxiliary screen, however, requires that the user look away from the application display, and hence destroys visual continuity. The advantages are the saving of display space, which is often at a premium, and the accommodation of a large set of commands in one menu.

A pop-up menu appears on the screen when a selection is to be made, either in response to an explicit user action (typically pressing a mouse or tablet puck button), or automatically because the next dialogue step requires a menu selection. The menu normally appears at the cursor location, which is usually the user's center of visual attention, thereby



Fig. 8.12 A dual-display workstation. The two displays can be used to show the overview of a drawing on one and detail on the other, or to show the drawing on one and menus on the other. (Courtesy of Intergraph Corporation, Huntsville, Al.)

Samsung/Google, EX1027, p. 383

maintaining visual continuity. An attractive feature in pop-up menus is to highlight initially the most recently made selection from the choice set *if* the most recently selected item is more likely to be selected a second time than is another item, positioning the menu so the cursor is on that item. Alternatively, if the menu is ordered by frequency of use, the most frequently used command can be highlighted initially and should also be in the middle (not at the top) of the menu, to minimize cursor movements in selecting other items.

Pop-up and other appearing menus conserve precious screen space—one of the user-interface designer's most valuable commodities. Their use is facilitated by a fast RasterOp instruction, as discussed in Chapters 2 and 19.

Pop-up menus often can be context-sensitive. In several window-manager systems, if the cursor is in the window banner (the top heading of the window), commands involving window manipulation appear in the menu; if the cursor is in the window proper, commands concerning the application itself appear (which commands appear can depend on the type of object under the cursor); otherwise, commands for creating new windows appear in the menu. This context-sensitivity may initially be confusing to the novice, but is powerful once understood.

Unlike pop-up menus, pull-down and pull-out menus are anchored in a menu bar along an edge of the screen. The Apple Macintosh, Microsoft Windows, and Microsoft Presentation Manager all use pull-down menus. Macintosh menus, shown in Fig 8.13, also illustrate accelerator keys and context sensitivity. Pull-out menus, an alternative to pull-down menus, are shown in Fig. 8.14. Both types of menus have a two-level hierarchy: The menu bar is the first level, and the pull-down or pull-out menu is the second. Pull-down and pull-out menus can be activated explicitly or implicitly. In explicit activation, a button depression, once the cursor is in the menu bar, makes the second-level menu appear; the

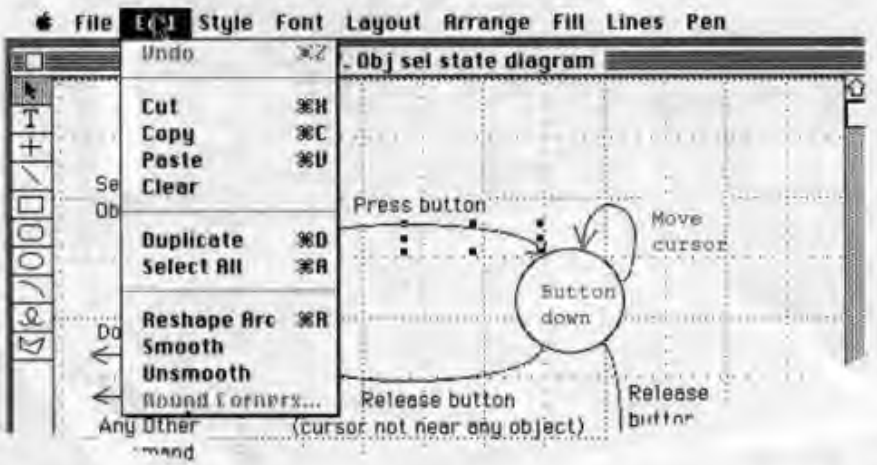


Fig. 8.13 A Macintosh pull-down menu. The last menu item is gray rather than black, indicating that it is currently not available for selection (the currently selected object, an arc, does not have corners to be rounded). The Undo command is also gray, because the previously executed command cannot be undone. Abbreviations are accelerator keys for power users. (Copyright 1988 Claris Corporation. All rights reserved.)

Samsung/Google, EX1027, p. 384

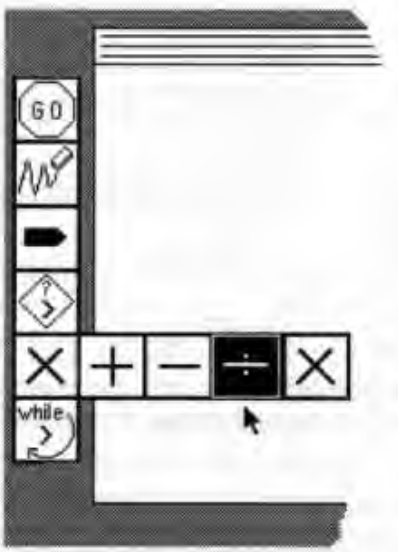


Fig. 8.14 A pull-out menu in which the leftmost, permanently displayed element shows the current selection. The newly selected menu item (reversed background) will become the current selection. This contrasts with most menu styles, in which the name of the menu is permanently displayed and the current selection is not shown after the adapted menu is dismissed. Menu is adapted from Jovanović's Process Visualization System [JOVA86]. (Courtesy of Branka Jovanović.)

cursor is moved on top of the desired selection and the button is then released. In implicit activation, moving the cursor into the heading causes the menu to appear; no button press is needed. Either selecting an entry or moving the cursor out of the menu area dismisses the menu. These menus, sometimes called "lazy" or "drop-down" menus, may also confuse new users by their seemingly mysterious appearance.

A full-screen menu can be a good or bad solution, depending on the context within which it is used. The disadvantage is that the application drawing will be obscured, removing context that might help the user to make an appropriate choice. Even this concern can be removed by using a raster display's look-up table to show the menu in a strong, bright color, over a dimmed application drawing [FEIN82a].

Visual representation. The basic decision on representation is whether menus use textual names or iconic or other graphical representations of elements of the choice set. Full discussion of this topic is deferred to the next chapter; however, note that iconic menus can be spatially organized in more flexible ways than can textual menus, because icons need not be long and thin like text strings; see Fig. 8.15. Also, inherently graphical concepts (particularly graphical attributes and geometrical primitives) are easily depicted.

Current selection. If a system has the concept of "currently selected element" of a choice set, menu selection allows this element to be highlighted. In some cases, an initial default setting is provided by the system and is used unless the user changes it. The currently selected element can be shown in various ways. The *radio-button* interaction

Samsung/Google, EX1027, p. 385

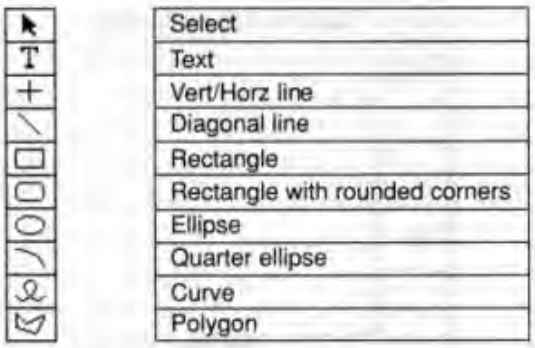


Fig. 8.15 Iconic and textual menus for the same geometric primitives. The iconic menu takes less space than does the textual menu. (Icons © 1988 Claris Corporation. All rights reserved.)

technique, patterned after the tuning buttons on car radios, is one way (Fig. 8.16). Again, some pop-up menus highlight the most recently selected item and place it under the cursor, on the assumption that the user is more likely to reselect that item than she is to select any other entry.

Size and shape of menu items. Pointing accuracy and speed are affected by the size of each individual menu item. Larger items are faster to select, as predicted by Fitts' law [FITT54; CARD83]; on the other hand, smaller items take less space and permit more menu items to be displayed in a fixed area, but induce more errors during selection. Thus, there is a conflict between using small menu items to preserve screen space versus using larger ones to decrease selection time and to reduce errors.

Pop-up *pie menus* [CALL88], shown in Fig. 8.17, appear at the cursor. As the user moves the mouse from the center of the pie toward the desired selection, the target width becomes larger, decreasing the likelihood of error. Thus, the user has explicit control over the speed-versus-error tradeoff. In addition, the distance to each menu item is the same.

Pattern recognition. In selection techniques involving pattern recognition, the user makes sequences of movements with a continuous-positioning device, such as a tablet or



Fig. 8.16 Radio-button technique for selecting from a set of mutually exclusive alternatives. (Courtesy of NeXT, Inc. © 1989 NeXT, Inc.)

Samsung/Google, EX1027, p. 386



Fig. 8.17 A four-element pie menu.

mouse. The pattern recognizer automatically compares the sequence with a set of defined patterns, each of which corresponds to an element of the selection set. Figure 8.18 shows one set of sketch patterns and their related commands, taken from Wallace's SELMA queueing analyzer [IRAN71]. Proofreader's marks indicating delete, capitalize, move, and so on are attractive candidates for this approach [WOLF87].

The technique requires no typing skill and preserves tactile continuity. Furthermore, if the command involves an object, the cursor position can be used for selection. The move command used in many Macintosh applications is a good example: the cursor is positioned on top of the object to be moved and the mouse button is pressed, selecting the object under

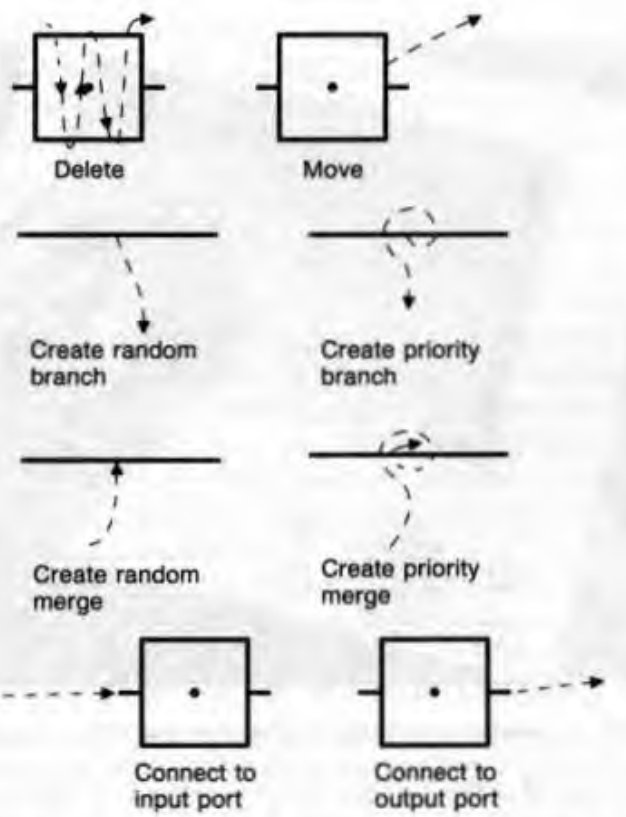


Fig. 8.18 Motions, indicated as dotted lines, that are recognized as commands. From Wallace's SELMA queuing analyzer [IRAN71].

Samsung/Google, EX1027, p. 387

the cursor (it is displayed in reverse video for feedback). As the user moves the mouse (still holding down the button), the object moves also. Releasing the mouse button detaches the object from the mouse. Skilled operators can work very rapidly with this technique, because hand movements between the work area and a command-entry device are eliminated. Given a data tablet and stylus, this technique can be used with at least several dozen patterns, but it is difficult for the user to learn a large number of different patterns.

Rhyne has recently combined a transparent tablet and liquid-crystal display into a prototype of a portable, lap-top computer [RHYN87]. Patterns are entered on the transparent tablet, and are recognized and interpreted as commands, numbers, and letters. The position at which information is entered is also significant. Figure 8.19 shows the device in use with a spreadsheet application.

Function keys. Elements of the choice set can be associated with function keys. (We can think of single-keystroke inputs from a regular keyboard as function keys.) Unfortunately, there never seem to be enough keys to go around! The keys can be used in a hierarchical-selection fashion, and their meanings can be altered using chords, say by depressing the keyboard shift and control keys along with the function key itself. Learning exotic key combinations, such as "shift-option-control-L," for some commands is not easy, however, and is left as an exercise for the regular user seeking the productivity gains that typically result. Putting a "cheat-sheet" template on the keyboard to remind users of



Fig. 8.19 An IBM experimental display and transparent data tablet. A spreadsheet application is executing. The user has just circled three numbers, and has indicated that their sum is to be entered in the cell containing the sigma. A new column heading has also been printed. The system recognizes the gestures and characters, and enters appropriate commands to the spreadsheet application. (Courtesy IBM T. J. Watson Research Laboratories.)

Samsung/Google, EX1027, p. 388

these obscure combinations can speed up the learning process. It is even sometimes possible to define chords that make some sense, decreasing learning time. For instance, Microsoft Word on the Macintosh uses "shift-option->" to increase point size and the symmetrical "shift-option-<" to decrease point size; "shift-option-I" italicizes plain text and unitalicizes italicized text, whereas "shift-option-U" treats underlined text similarly.

One way to compensate for the lack of multiple buttons on a mouse is to use the temporal dimension to expand the possible meanings of one button—for instance, by distinguishing between a single click and two clicks made in rapid succession. If the meaning of two clicks is logically related to that of one click, this technique can be especially effective; otherwise, rote memory is needed to remember what one and two clicks mean. Examples of this technique are common: one click on a file icon selects it; two clicks opens the file. One click on the erase command enters erase mode; two clicks erase the entire screen. This technique can also be applied to each of the buttons on a multibutton mouse. Chording of mouse buttons or of keyboard keys with mouse buttons can also be used to provide the logical (but not necessarily human-factors) equivalent of more buttons. To be most useful, the organizing scheme for the chording patterns must be logical and easy to remember.

8.2.4 The Text Interaction Task

The text-string input task entails entering a character string to which the application does not ascribe any special meaning. Thus, typing a command name is *not* a text-entry task. In contrast, typing legends for a graph and typing text into a word processor *are* text input tasks. Clearly, the most common text-input technique is use of the QWERTY keyboard.

Character recognition. The user prints characters with a continuous-positioning device, usually a tablet stylus, and the computer recognizes them. This is considerably easier than recognizing scanned-in characters, because the tablet records the sequence, direction, and sometimes speed and pressure of strokes, and a pattern-recognition algorithm can match these to stored templates for each character. For instance, the capital letter "A" consists of three strokes—typically, two downward strokes and one horizontal stroke. A recognizer can be trained to identify different styles of block printing: the parameters of each character are calculated from samples drawn by the user. Character recognizers have been used with interactive graphics since the early 1960s [BROW64; TEIT64]. A simplified adaptation of Teitelman's recognizer, developed by Ledeen, is described in [NEWM73]; a commercial system is described in [WARD85; BLES86].

It is difficult to block print more than one or two characters per second (try it!), so character recognition is not appropriate for massive input of text. We write cursive letters faster than we print the same characters, but there are no simple recognition algorithms for cursive letters: the great variability among individuals' handwrtiting and the difficulty of segmenting words into individual letters are two of the problems.

Menu selection. A series of letters, syllables, or other basic units is displayed as a menu. The user then inputs text by choosing letters from the menu with a selection device. This technique is attractive in several situations. First, if only a short character string is to be entered and the user's hands are already on a pointing device, then menu selection may be

Samsung/Google, EX1027, p. 389

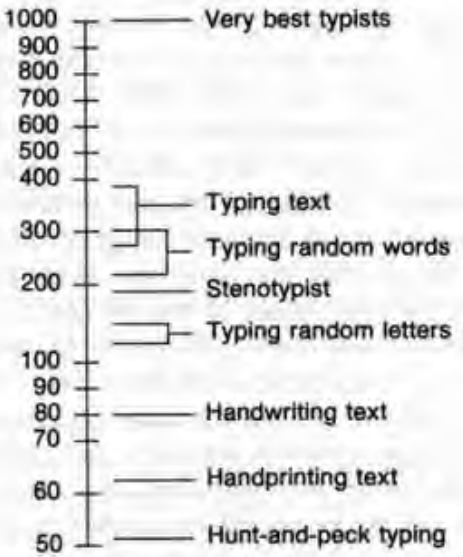


Fig. 8.20 Data-input speeds, in keystrokes per minute, of various techniques for entering text and numeric information. (Adapted from [VANC72, p. 335] and [CARD83, p. 61].)

faster than moving to the keyboard and back. Second, if the character set is large, this approach is a reasonable alternative to the keyboard.

Hierarchical menu selection can also be used with large character sets, such as are used in Chinese and Japanese. One such system uses the graphical features (strong horizontal line, strong vertical line, etc.) of the symbols for the hierarchy. A more common strategy is to enter the word in phonetic spelling, which string is then matched in a dictionary. For example, the Japanese use two alphabets, the katakana and hiragana, to type phonetically the thousands of kanji characters that their orthography borrows from the Chinese.

Evaluation of text-entry techniques. For massive input of text, the only reasonable substitute for a skilled typist working with a keyboard is an automatic scanner. Figure 8.20 shows experimentally determined keying rates for a variety of techniques. The hunt-and-peck typist is slowed by the perceptual task of finding a key and the ensuing motor task of moving to and striking it, but the trained typist has only the motor task of striking the key, preceded sometimes by a slight hand or finger movement to reach it. Speech input, not shown on the chart, is slow but attractive for applications where the hands must be free for other purposes, such as handling paperwork.

8.2.5 The Quantify Interaction Task

The quantify interaction task involves specifying a numeric value between some minimum and maximum value. Typical interaction techniques are typing the value, setting a dial to the value, and using an up–down counter to select the value. Like the positioning task, this task may be either linguistic or spatial. When it is linguistic, the user knows the specific value to be entered; when it is spatial, the user seeks to increase or decrease a value by a

Samsung/Google, EX1027, p. 390

certain amount, with perhaps an approximate idea of the desired end value. In the former case, the interaction technique clearly must involve numeric feedback of the value being selected (one way to do this is to have the user type the actual value); in the latter case, it is more important to give a general impression of the approximate setting of the value. This is typically accomplished with a spatially oriented feedback technique, such as display of a dial or gauge on which the current (and perhaps previous) value is shown.

One means of entering values is the potentiometer. The decision of whether to use a rotary or linear potentiometer should take into account whether the visual feedback of changing a value is rotary (e.g., a turning clock hand) or linear (e.g., a rising temperature gauge). The current position of one or a group of slide potentiometers is much more easily comprehended at a glance than are those of rotary potentiometers, even if the knobs have pointers; unfortunately, most graphics system manufacturers offer only rotary potentiometers. On the other hand, rotary potentiometers are easier to adjust. Availability of both linear and rotary potentiometers can help users to associate meanings with each device. It is important to use directions consistently: clockwise or upward movements normally increase a value.

With continuous-scale manipulation, the user points at the current-value indicator on a displayed gauge or scale, presses the selection button, drags the indicator along the scale to the desired value, and then releases the selection button. A pointer is typically used to indicate the value selected on the scale, and a numeric echo may be given. Figure 8.21 shows several such dials and their associated feedback. The range or precision of values entered in this way can be extended by using the positioning device as a relative rather than absolute device and by using a nonconstant C/D ratio, as discussed in Section 8.1.1. Then it becomes possible to increase a value by repeating a series of stroking actions: move to the right, lift mouse, move to the left, put mouse down, and so on. Thornton's number wheel [THOR79] is such a technique.

If the resolution needed is higher than the continuous-scale manipulation technique can provide, or if screen space is at a premium, an up–down counter arrangement can be used, as shown in Fig. 8.22.

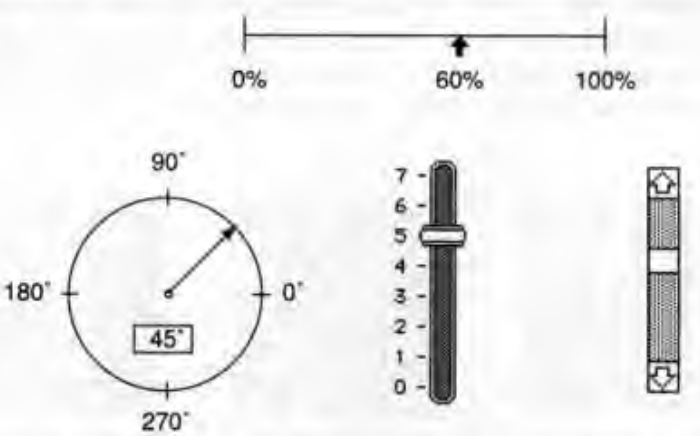


Fig. 8.21 Several dials that the user can use to input values by dragging the control pointer. Feedback is given by the pointer and, in two cases, by numeric displays. (Vertical sliders © Apple Computer, Inc.)

Samsung/Google, EX1027, p. 391

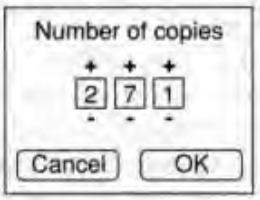


Fig. 8.22 An up–down counter for specifying a value. The user positions the cursor on "+" or "−" and holds down the selection button on the pointing device; the corresponding digit increases or decreases until the button is released.

8.2.6 3D Interaction Tasks

Two of the four interaction tasks described previously for 2D applications become more complicated in 3D: position and select. In this section, we also introduce an additional 3D interaction task: rotate (in the sense of orienting an object in 3-space). The major reason for the complication is the difficulty of perceiving 3D depth relationships of a cursor or object relative to other displayed objects. This contrasts starkly with 2D interaction, where the user can readily perceive that the cursor is above, next to, or on an object. A secondary complication arises because the commonly available interaction devices, such as mice and tablets, are only 2D devices, and we need a way to map movements of these 2D devices into 3D.

Display of stereo pairs, corresponding to left- and right-eye views, is helpful for understanding general depth relationships, but is of limited accuracy as a precise locating method. Methods for presenting stereo pairs to the eye are discussed in Chapters 14 and 18, and in [HODG85]. Other ways to show depth relationships are discussed in Chapters 14–16.

The first part of this section deals with techniques for positioning and selecting, which are closely related. The second part concerns techniques for interactive rotation.

Figure 8.23 shows a common way to position in 3D. The 2D cursor, under control of, say, a mouse, moves freely among the three views. The user can select any one of the 3D cursor's dashed lines and can drag the line, using a button-down–drag–button-up sequence. If the button-down event is close to the intersection of two dashed cursor lines, then both are selected and are moved with the mouse (gravity, discussed in Section 8.3.2, can make



Fig. 8.23 3D positioning technique using three views of the same scene (a house). The 2D cursor (+) is used to select one of the dashed 3D cursor lines.

Samsung/Google, EX1027, p. 392

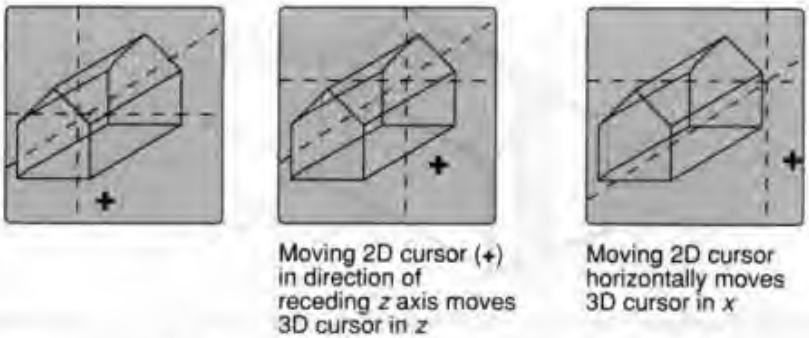


Fig. 8.24 Movement of the 3D cursor is controlled by the direction in which the 2D cursor is moved.

picking the intersection especially easy). Although this method may appear restrictive in forcing the user to work in one or two dimensions at a time, it is sometimes advantageous to decompose the 3D manipulation task into simpler lower-dimensional tasks. Selecting as well as locating is facilitated with multiple views: Objects that overlap and hence are difficult to distinguish in one view may not overlap in another view.

Another possibility, developed by Nielson and Olsen [NIEL86] and depicted in Fig. 8.24, requires that all three principal axes project with nonzero length. A 3D cross-hair cursor, with cross-hairs parallel to the principal axes, is controlled by moving the mouse in the general direction of the projections of the three principal axes. Figure 8.25 shows how 2D locator movements are mapped into 3D: there are 2D zones in which mouse movements affect a specific axis. Of course, 3D movement is restricted to one axis at a time.

Both of these techniques illustrate ways to map 2D locator movements into 3D movements. We can instead use buttons to control which of the 3D coordinates are affected by the locator's 2 degrees of freedom. For example, the locator might normally control x and y; but, with the keyboard shift key depressed, it could control x and z instead (this requires that the keyboard be unencoded, as discused in Chapter 4). Alternatively, three buttons could be used, to bind the locator selectively to an axis. Instead of mapping from a

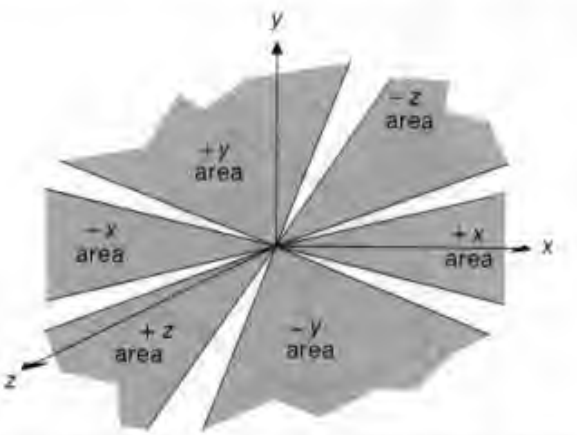


Fig. 8.25 The six regions of mouse movement, which cause the 3D cursor to move along the principal axes.

Samsung/Google, EX1027, p. 393

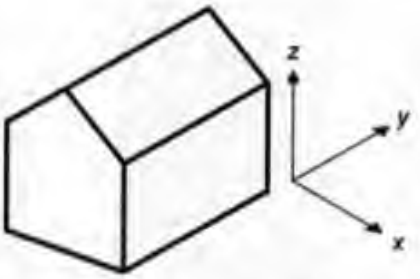


Fig. 8.26 The displayed local coordinate system of the house, which shows the three directions in which any translated object will move. To preserve stimulus-response compatibility, we can use the direction of mouse movements to determine the axes chosen, as in Fig. 8.25.

2D device into 3D, we could use a real 3D locator, such as the joysticks and trackballs discussed in Section 8.1.6

Constrained 3D movement is effective in 3D locating. Gridding and gravity can sometimes compensate for uncertainties in depth relationships and can aid exact placement. Another form of constraint is provided by those physical devices that make it easier to move along principal axes than in other directions. Some trackballs and joysticks have this property, which can also be simulated with the isometric strain-gauge and spaceball devices (Section 8.1.6).

Context-specific constraints are often more useful, however, than are these general constraints. It is possible to let the user specify that movements should be parallel to or on lines or planes other than the principal axes and planes. For example, with a method developed by Nielson and Olsen [NIEL86], the local coordinate system of the selected object defines the directions of movement as shown in Fig. 8.26. In a more general technique developed by Bier [BIER86b], the user places a coordinate system, called a *skitter*, on the surface of an object, again defining the possible directions of movement (Fig. 8.27). Another way to constrain movement to a particular plane is to give the user control over the view-plane orientation, and to limit translation to be parallel to the view plane.

One method of 3D picking—finding the output primitive that, for an (x, y) position

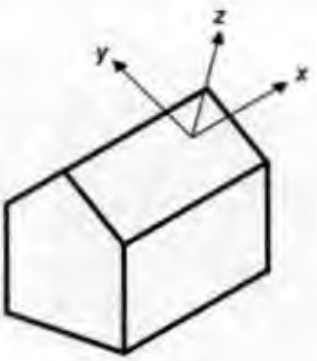


Fig. 8.27 The displayed coordinate system, placed interactively so that its (x, y) plane coincides with the plane of the roof, shows the three directions in which any translated object will move.

Samsung/Google, EX1027, p. 394

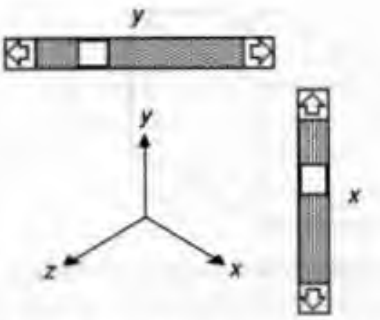


Fig. 8.28 Two slider dials for effecting rotation about the screen x and y axes.

determined by a 2D locator, has the maximum z value—was discussed in Chapter 7. Another method, which can be used with a 3D locator when wireframe views are shown—is to find the ouput primitive closest to the locator's (x, y, z) position.

As with locating and selection, the issues in 3D rotation are understanding depth relationships, mapping 2D interaction devices into 3D, and ensuring stimulus-response compatibility. An easily implemented 3D rotation technique provides slider dials or gauges that control rotation about three axes. S-R compatibility suggests that the three axes should normally be in the screen-coordinate system—x to the right, y increasing upward, z out of (or into) the screen [BRIT78]. Of course, the center of rotation either must be explicitly specified as a separate step, or must be implicit (typically the screen-coordinate origin, the origin of the object, or the center of the object). Providing rotation about the sceen's x and y axes is especially simple, as suggested in Fig. 8.28. The (x, y, z) coordinate system associated with the sliders is rotated as the sliders are moved to show the effect of the rotation. A 2D trackball can be used instead of the two sliders.

The two-axis rotation approach can be easily generalized to three axes by adding a dial for z-axis rotation, as in Fig. 8.29 (a dial is preferable to a slider for S-R compatibility).

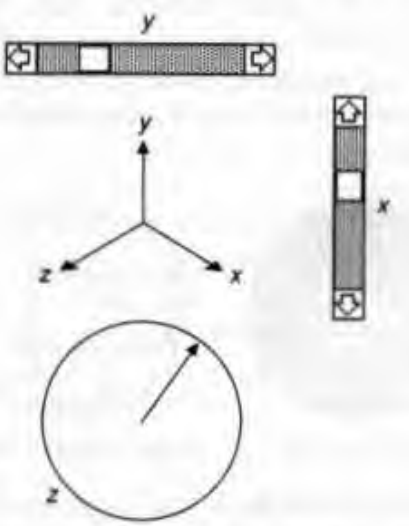


Fig. 8.29 Two slider dials for effecting rotation about the screen x and y axes, and a dial for rotation about the screen z axis. The coordinate system represents world coordinates and shows how world coordinates relate to screen coordinates.

Samsung/Google, EX1027, p. 395

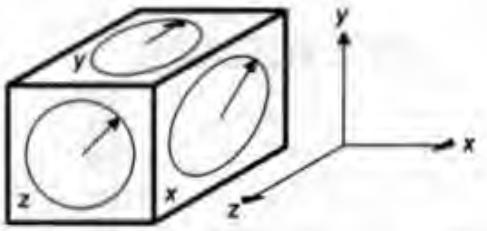
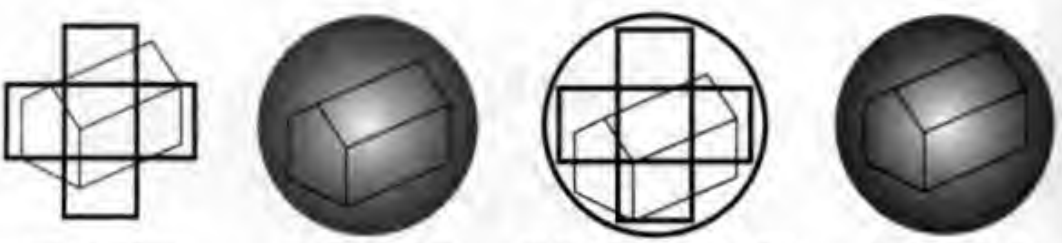


Fig. 8.30 Three dials to control rotation about three axes. The placement of the dials on the cube provides strong stimulus-response compatibility.

Even more S-R compatibility comes from the arrangement of dials on the faces of a cube shown in Fig. 8.30, which clearly suggests the axes controlled by each dial. Again, a 3D trackball could be used instead of the dials.

Mouse movements can be directly mapped onto object movements, without slider or dial intermediaries. The user can be presented a metaphor in which the two sliders of Fig. 8.28 are superimposed on top of the object being rotated, so that horizontal mouse movements are mapped into rotations about the screen-coordinate y axis, and vertical mouse movements are mapped into rotations about the screen-coordinate x axis (Fig. 8.31a). Diagonal motions have no effect. The slider dials are not really displayed; the user learns to imagine that they are present. Alternatively, the user can be told that an imaginary 2D trackball is superimposed on top of the object being rotated, so that the vertical, horizontal, or diagonal motions one would make with the trackball can be made instead with the mouse (Fig. 8.31b). Either of these methods provides two-axis rotation in 3D.

For three-axis rotations, three methods that closely resemble real-world concepts are particularly interesting. In the overlapping-sliders method [CHEN88], the user is shown two linear sliders overlapping a rotary slider, as in Fig. 8.31(c). Motions in the linear sliders control rotation about the x and y axes, while a rotary motion around the intersection of the two linear sliders controls rotation about the z axis. In a technique developed by Evans, Tanner, and Wein [EVAN81], three successive mouse positions are compared to determine whether the mouse motion is linear or rotary. Linear horizontal or vertical movements control rotation about the x and y axes, a linear diagonal movement rotates



(a) Two sliders (b) 2D trackball (c) Two sliders, rotary dial (d) 3D trackball

Fig. 8.31 Four methods of 3D rotation. In each case, the user makes movements with a 2D device corresponding to those that would be made if the actual devices were superimposed on the object. A 3D trackball can be twisted to give z-axis rotation, whereas a 2D trackball provides only two-axis rotation.

Samsung/Google, EX1027, p. 396

about both the x and y axes, and rotary movements control rotation about the z axis. While this is a relative technique and does not require that the movements be made directly over the object being rotated or in a particular area, the user can be instructed to use these motions to manipulate a 3D trackball superimposed on the object (Fig. 8.31d). In the virtual-sphere method, also developed by Chen [CHEN88], the user actually manipulates this superimposed 3D trackball in an absolute fashion as though it were real. With a mouse button down, mouse movements rotate the trackball exactly as your finger would move a real trackball. An experiment [CHEN88] comparing these latter two approaches showed no performance differences, but did yield a user preference for Chen's method.

It is often necessary to combine 3D interaction tasks. Thus, rotation requires a select task for the object to be rotated, a position task for the center of rotation, and an orient task for the actual rotation. Specifying a 3D view can be thought of as a combined positioning (where the eye is), orientation (how the eye is oriented), and scaling (field of view, or how much of the projection plane is mapped into the viewport) task. We can create such a task by combining some of the techniques we have discussed, or by designing a *fly-around* capability in which the viewer flies an imaginary airplane around a 3D world. The controls are typically pitch, roll, and yaw, plus velocity to speed up or slow down. With the fly-around concept, the user needs an overview, such as a 2D plan view, indicating the imaginary airplane's ground position and heading.

8.3 COMPOSITE INTERACTION TASKS

Composite interaction tasks (CITs) are built on top of the basic interaction tasks (BITs) described in the previous section, and are actually combinations of BITs integrated into a unit. There are three major forms of CITs: dialogue boxes, used to specify multiple units of information; construction, used to create objects requiring two or more positions; and manipulation, used to reshape existing geometric objects.

8.3.1 Dialogue Boxes

We often need to select multiple elements of a selection set. For instance, text attributes, such as italic, bold, underline, hollow, and all caps, are not mutually exclusive, and the user may want to select two or more at once. In addition, there may be several sets of relevant attributes, such as typeface and font. Some of the menu approaches useful in selecting a single element of a selection set are not satisfactory for multiple selections. For example, pull-down and pop-up menus normally disappear when a selection is made, necessitating a second activation to make a second selection.

This problem can be overcome with dialogue boxes, a form of menu that remains visible until explicitly dismissed by the user. In addition, dialogue boxes can permit selection from more than one selection set, and can also include areas for entering text and values. Selections made in a dialogue box can be corrected immediately. When all the information has been entered into the dialogue box, the box is typically dismissed explicitly with a command. Attributes and other values specified in a dialogue box can be applied immediately, allowing the user to preview the effect of a font or line-style change. An "apply" command is sometimes included in the box to cause the new values to be used

Samsung/Google, EX1027, p. 397

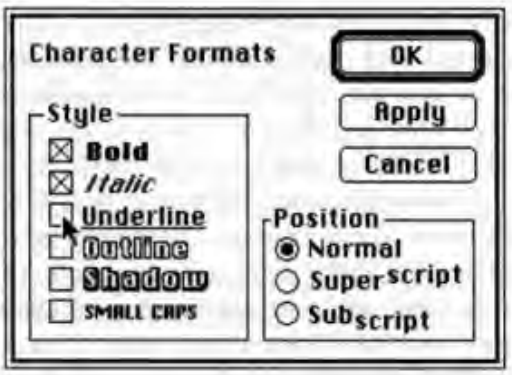


Fig. 8.32 A text-attribute dialogue box with several different attributes selected. The bold border of the "OK" button shows that the keyboard return key can be used as an alternative. The "Apply" button is used to apply new attibute values, so the user can observe their effects. "Cancel" is used to revert any changes to their previous values. Note that text attributes are described both by name and, graphically, by example. (Screen shots © 1983–1989 Microsoft Corporation. Reprinted with permission from Microsoft Corporation.)

without dismissing the box. More frequently, however, the dialogue box must be dismissed to apply the new settings. Figure 8.32 shows a dialogue box with several selected items highlighted.

8.3.2 Construction Techniques

One way to construct a line is to have the user indicate one endpoint and then the other; once the second endpoint is specified, a line is drawn between the two points. With this technique, however, the user has no easy way to try out different line positions before settling on a final one, because the line is not actually drawn until the second endpoint is given. With this style of interaction, the user must invoke a command each time an endpoint is to be repositioned.

A far superior approach is *rubberbanding*, discussed in Chapter 2. When the user pushes a button (often the tipswitch on a tablet stylus, or a mouse button), the starting position of the line is established by the cursor (usually but not necessarily controlled by a continuous-positioning device). As the cursor moves, so does the endpoint of the line; when the button is released, the endpoint is frozen. Figure 8.33 show a rubberband line-drawing sequence. The user-action sequence is shown in the state diagram in Fig. 8.34. Notice that the state "rubberband" is active *only* while a button is held down. It is in this state that cursor movements cause the current line to change. See [BUXT85] for an informative discussion of the importance of matching the state transitions in an interaction technique with the transitions afforded by the device used with the technique.

An entire genre of interaction techniques is derived from rubberband line drawing. The *rubber-rectangle* technique starts by anchoring one corner of a rectangle with a button-

Samsung/Google, EX1027, p. 398

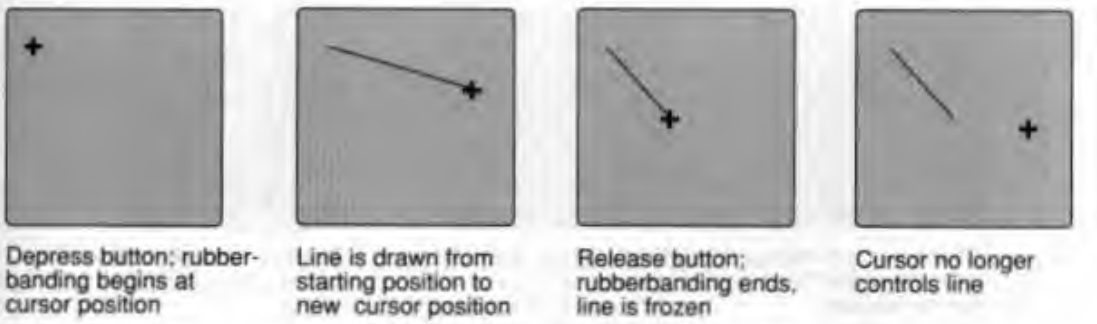


Fig. 8.33 Rubberband line drawing.

down action, after which the opposite corner is dynamically linked to the cursor until a button-up action occurs. The state diagram for this technique differs from that for rubberband line drawing only in the dynamic feedback of a rectangle rather than a line. The *rubber-circle* technique creates a circle that is centered at the initial cursor position and that passes through the current cursor position, or that is within the square defined by opposite corners. The *rubber-ellipse* technique creates an axis-aligned ellipse inside the rectangle defined by the initial and current cursor positions. A circle results if the rectangle is square—easily done with gridding. All these techniques have in common the user-action sequence of button-down, move locator and see feedback, button-up.

One interaction technique to create a polyline (a sequence of connected lines) is an extension of rubberbanding. After entering the polyline-creation command, the user clicks on a button to anchor each rubberbanded vertex. After all the vertices have been indicated, the user indicates completion, typically by a double click on a button without moving the locator, by a click on a second mouse button, or by entry of a new command. If the new command is from a menu, the last line segment of the polyline follows the cursor to the menu, and then disappears. Figure 8.35 depicts a typical sequence of events in creating a polyline; Fig. 8.36 is the accompanying state diagram.

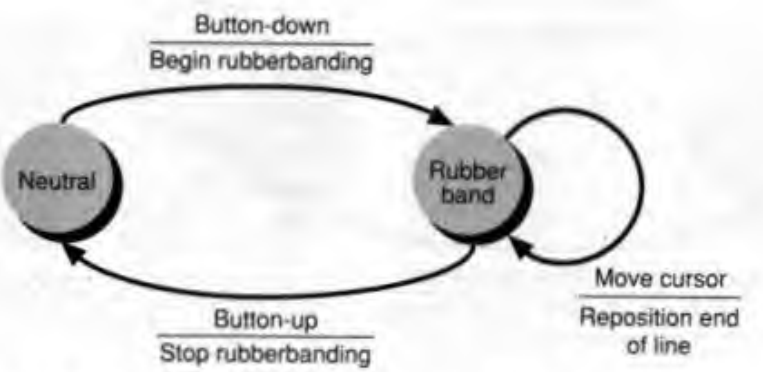


Fig. 8.34 State diagram for rubberband line drawing.

Samsung/Google, EX1027, p. 399

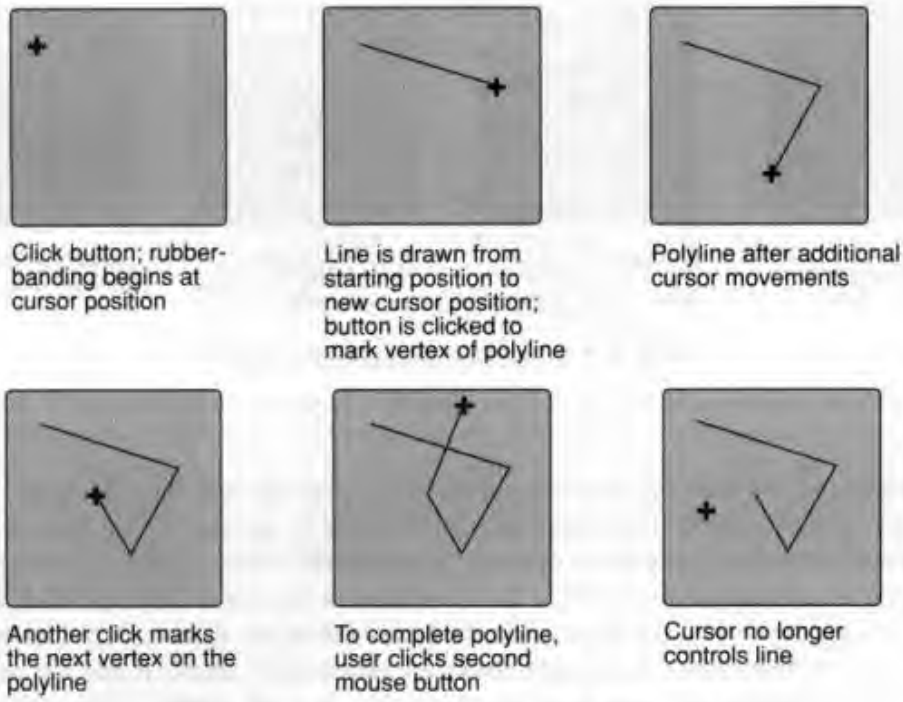


Fig. 8.35 Rubberband polyline sketching.

A polygon can be drawn similarly. In some cases, the user signals to the system that the polygon is complete by returning the cursor to the starting vertex of the polygon. In other cases, the user explicitly signals completion using a function key, and the system automatically closes the polygon. Figure 8.37 shows one way to create polygons.

Constraints of various types can be applied to the cursor positions in any of these techniques. For example, Fig. 8.38 shows a sequence of lines drawn using the same cursor

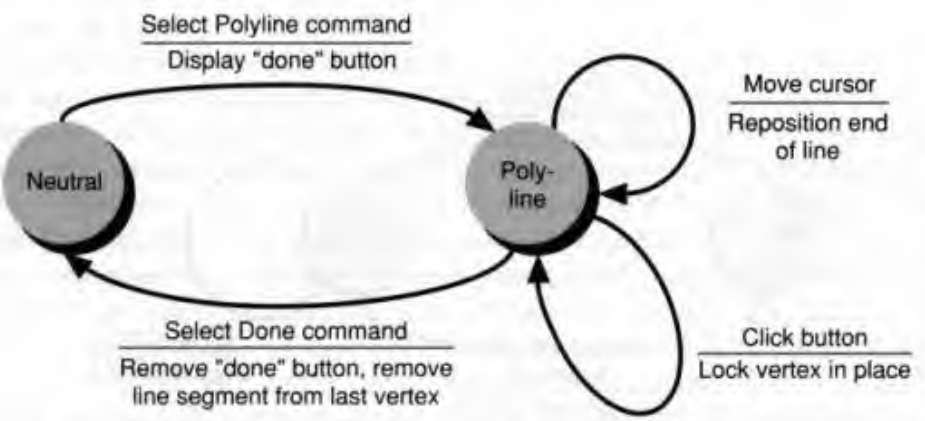


Fig. 8.36 State diagram for rubberband creation of a polyline.

Samsung/Google, EX1027, p. 400

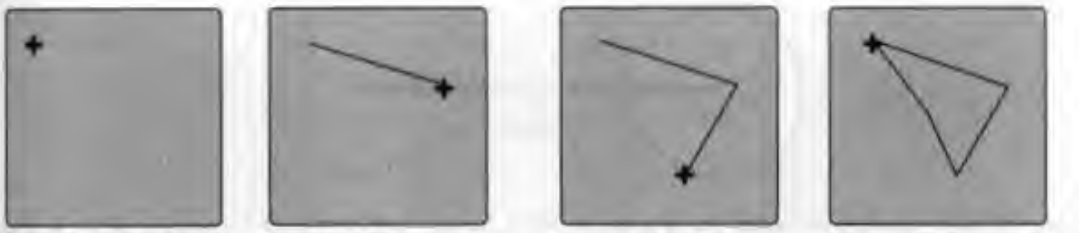


Fig. 8.37 Rubberband drawing of a polygon.

positions as in Fig. 8.33, but with a horizontal constraint in effect. A vertical line, or a line at some other orientation, can also be drawn in this manner. Polylines made entirely of horizontal and vertical lines, as in printed circuit boards, VLSI chips, and some city maps, are readily created; right angles are introduced either in response to a user command, or automatically as the cursor changes direction. The idea can be generalized to any shape, such as a circle, ellipse, or any other curve; the curve is initialized at some position, then cursor movements control how much of the curve is displayed. In general, the cursor position is used as input to a constraint function whose output is then used to display the appropriate portion of the object.

Gravity is yet another form of constraint. When constructing drawings, we frequently want a new line to begin at the endpoint of, or on, an existing line. Matching an endpoint is easy if it was created using gridding, but otherwise is difficult without a potentially time-consuming zoom. The difficulty is avoided by programming an imaginary gravity field around each existing line, so that the cursor is attracted to the line as soon as it enters the gravity field. Figure 8.39 shows a line with a gravity field that is larger at the endpoints, so that matching endpoints is especially easy.

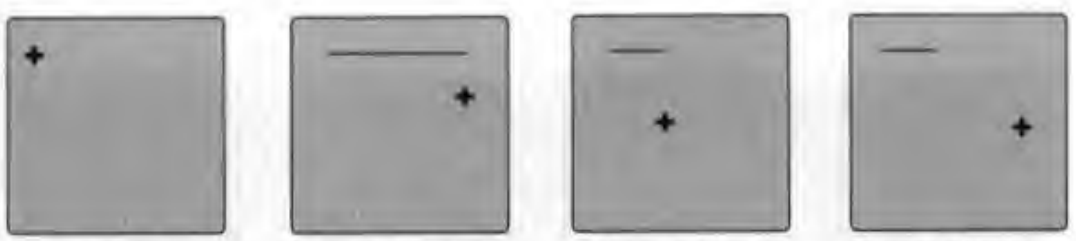


Fig. 8.38 Horizontally constrained rubberband line drawing.

Samsung/Google, EX1027, p. 401

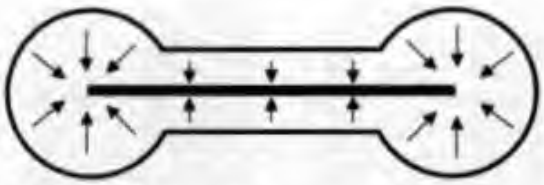


Fig. 8.39 Line surrounded by a gravity field, to aid picking points on the line: If the cursor falls within the field, it is snapped to the line.

8.3.3 Dynamic Manipulation

It is not sufficient to create lines, rectangles, and so on. In many situations, the user must be able to modify previously created geometric entities.

Dragging moves a selected symbol from one position to another under control of a cursor, as in Fig. 8.40. A button-down action typically starts the dragging (in some cases, the button-down is also used to select the symbol under the cursor to be dragged); then, a button-up freezes the symbol in place, so that further movements of the cursor have no effect on it. This button-down–drag–button-up sequence is often called *click-and-drag* interaction.

Dynamic rotation of an object can be done in a similar way, except that we must be able to identify the point or axis about which the rotation is to occur. A convenient strategy is to have the system show the current center of rotation and to allow the user to modify it as desired. Figure 8.41 shows one such scenario. Note that the same approach can be used for scaling, with the center of scaling, rather than that of rotation, being specified by the user.

The concept of *handles* is useful to provide scaling of an object, without making the user think explicitly about where the center of scaling is. Figure 8.42 shows an object with eight handles, which are displayed as small squares at the corners and on the sides of the imaginary box surrounding the object. The user selects one of the handles and drags it to scale the object. If the handle is on a corner, then the corner diagonally opposite is locked in place. If the handle is in the middle of a side, then the opposite side is locked in place.

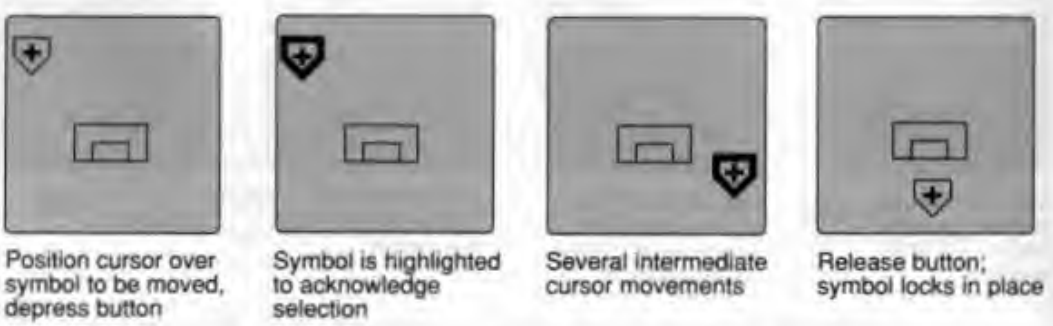


Fig. 8.40 Dragging a symbol into a new position.

Samsung/Google, EX1027, p. 402

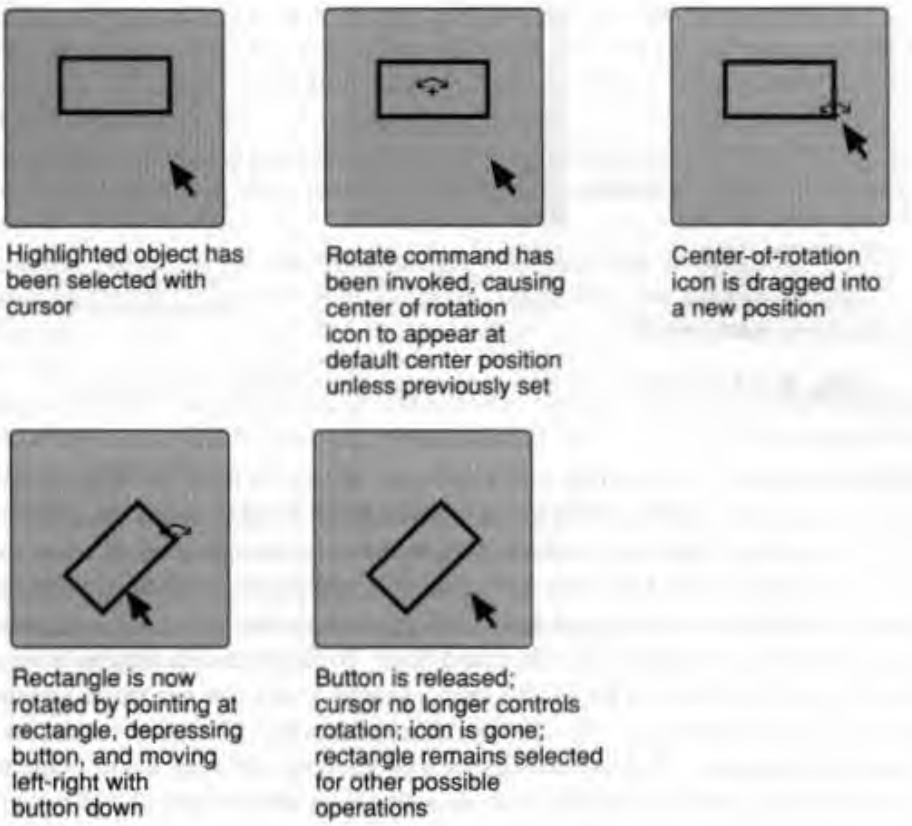


Fig. 8.41 Dynamic rotation.

When this technique is integrated into a complete user interface, the handles appear only when the object is selected to be operated on. Handles are also a unique visual code to indicate that an object is selected, since other visual codings (e.g., line thickness, dashed lines, or changed intensity) might also be used as part of the drawing itself. (Blinking is another unique visual code, but tends to be distracting and annoying.)

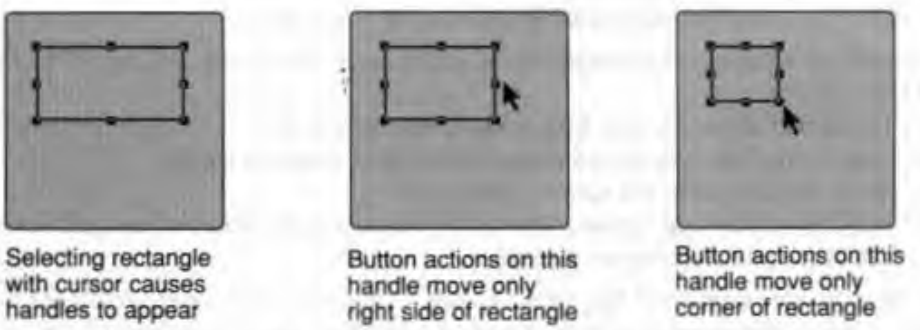


Fig. 8.42 Handles used to reshape objects.

Samsung/Google, EX1027, p. 403



Polygon has been selected for vertex modification, handle appears on each vertex

Depress-move-release over vertex causes vertex to move

Polygon no longer selected; handles have been removed

Fig. 8.43 Handles used to reposition the vertices of a polygon.

Dragging, rotating, and scaling affect an entire object. What if we wish to be able to move individual points, such as the vertices of a polygon? Vertices could be named, and the user could enter the name of a vertex and its new (x, y) coordinates. But the same point-and-drag strategy used to move an entire object is more attractive. In this case, the user points to a vertex, selects it, and drags it to a new position. The vertices adjacent to the one selected remain connected via rubberband lines. To facilitate selecting a vertex, we can establish a gravity field to snap the cursor onto a nearby vertex, we can make a vertex blink whenever the cursor is near, or we can superimpose handles over each vertex, as in Fig. 8.43. Similarly, the user can move an edge of a polygon by selecting it and dragging, with the edge maintaining its original slope. For smooth curves and surfaces, handles can also be provided to allow the user to manipulate points that control the shape, as discussed further in Chapter 11.

In the next chapter, we discuss design issues involved in combining basic and composite interaction techniques into an overall user–computer dialogue.

EXERCISES

8.1 Examine a user–computer interface with which you are familiar. List each interaction task used. Categorize each task into one of the four BITs of Section 8.2. If an interaction does not fit this classification scheme, try decomposing it further.

8.2 Implement adaptive C/D ratio cursor tracking for use with a mouse or other relative-positioning device. Experiment with different relationships between mouse velocity v and the C/D ratio r: $r = k\,v$ and $r = k\,v^2$. You must also find a suitable value for the constant k.

8.3 Conduct an experiment to compare the selection speed and accuracy of any of the following pairs of techniques:

a. Mouse and tablet selecting from a static, onscreen menu
b. Touch-panel and light-pen selecting from a static, onscreen menu,
c. Wide, shallow menu and narrow, deep menu
d. Pull-down menus that appear as soon as the cursor is in the menu bar, and pull-down menus that require a mouse-button depression.

8.4 Extend the state diagram of Fig. 8.6 to include a "return to lowest level" command that takes the selection back to the lowest level of the hierarchy, such that whatever was selected first is selected again.

Samsung/Google, EX1027, p. 404

8.5 Implement an autocompletion text-entry technique to use with an arbitrary list of words. Experiment with different word sets, such as the UNIX commands and proper names. Decide how to handle nonexistent matches, corrections typed by the user after a match has been made, and prompting for the user.

8.6 Implement cascading and panel heirarchical menus for a series of commands or for file-system subdirectories. What issues arise as you do this? Informally compare the selection speeds of each technique.

8.7 Implement pop-up menus that allow multiple selections prior to dismissal, which the user accomplishes by moving the cursor outside the menu. Alternatively, use a button click for dismissal. Which dismissal method do you prefer? Explain your answer. Ask five people who use the two techniques which dismissal method they prefer.

8.8 Implement a menu package on a color raster display that has a look-up table such that the menu is displayed in a strong, bright but partially transparent color, and all the colors underneath the menu are changed to a subdued gray.

8.9 Implement any of the 3D interaction techniques discussed in this chapter.

8.10 For each of the locating techniques discussed in Section 8.2.6, identify the line or plane into which 2D locator movements are mapped.

8.11 Draw the state diagram that controls pop-up hierarchical menus. Draw the state diagram that controls panel hierarchical menus.

Samsung/Google, EX1027, p. 405

9
Dialogue Design

We have described the fundamental building blocks from which the interface to an interactive graphics system is crafted—interaction devices, techniques, and tasks. Let us now consider how to assemble these building blocks into a usable and pleasing form. *User-interface design* is still at least partly an art, not a science, and thus some of what we offer is an attitude toward the design of interactive systems, and some specific dos and don'ts that, if applied creatively, can help to focus attention on the *human factors*, also called the *ergonomics*, of an interactive system.

The key goals in user-interface design are increase in speed of learning, and in speed of use, reduction of error rate, encouragement of rapid recall of how to use the interface, and increase in attractiveness to potential users and buyers.

Speed of learning concerns how long a new user takes to achieve a given proficiency with a system. It is especially important for systems that are to be used infrequently by any one individual: Users are generally unwilling to spend hours learning a system that they will use for just minutes a week!

Speed of use concerns how long an experienced user requires to perform some specific task with a system. It is critical when a person is to use a system repeatedly for a significant amount of time.

The *error rate* measures the number of user errors per interaction. The error rate affects both speed of learning and speed of use; if it is easy to make mistakes with the system, learning takes longer and speed of use is reduced because the user must correct any mistakes. However, error rate must be a separate design objective for applications in which even one error is unacceptable—for example, air-traffic control, nuclear-power-plant

Samsung/Google, EX1027, p. 406

control, and strategic military command and control systems. Such systems often trade off some speed of use for a lower error rate.

Rapid recall of how to use the system is another distinct design objective, since a user may be away from a system for weeks, and then return for casual or intensive use. The system should "come back" quickly to the user.

Attractiveness of the interface is a real marketplace concern. Of course, liking a system or a feature is not necessarily the same as being facile with it. In numerous experiments comparing two alternative designs, subjects state a strong preference for one design but indeed perform faster with the other.

It is sometimes said that systems cannot be both easy to learn and fast to use. Although there was certainly a time when this was often true, we have learned how to satisfy multiple design objectives. The simplest and most common approach to combining speed of use and ease of learning is to provide a "starter kit" of basic commands that are designed for the beginning user, but are only a subset of the overall command set. This starter kit is made available from menus, to facilitate ease of learning. All the commands, both starter and advanced, are available through the keyboard or function keys, to facilitate speed of use. Some advanced commands are sometimes put in the menus also, typically at lower levels of hierarchy, where they can be accessed by users who do not yet know their keyboard equivalents.

We should recognize that speed of learning is a relative term. A system with 10 commands is faster to learn than is one with 100 commands, in that users will be able to understand what each of the 10 commands does more quickly than they can what 100 do. But if the application for which the interface is designed requires rich functionality, the 10 commands may have to be used in creative and imaginative ways that are difficult to learn, whereas the 100 commands may map quite readily onto the needs of the application.

In the final analysis, meeting even one of these objectives is no mean task. There are unfortunately few absolutes in user-interface design. Appropriate choices depend on many different factors, including the design objectives, user characteristics, the environment of use, available hardware and software resources, and budgets. It is especially important that the user-interface designer's ego be submerged, so that the user's needs, not the designer's, are the driving factor. There is no room for a designer with quick, off-the-cuff answers. Good design requires careful consideration of many issues and patience in testing prototypes with real users.

9.1 THE FORM AND CONTENT OF USER–COMPUTER DIALOGUES

The concept of a *user–computer dialogue* is central to interactive system design, and there are helpful analogies between user–computer and person–person dialogues. After all, people have developed effective ways of communicating, and it makes sense to learn what we can from these years of experience. Dialogues typically involve gestures and words: In fact, people may have communicated with gestures, sounds, and images (cave pictures, Egyptian hieroglyphics) even before phonetic languages were developed. Computer graphics frees us from the limitations of purely verbal interactions with computers and enables us to use images as an additional communication modality.

Samsung/Google, EX1027, p. 407

Many attributes of person–person dialogues should be preserved in user–computer dialogues. People who communicate effectively share common knowledge and a common set of assumptions. So, too, there should be commonality between the user and the computer. Further, these assumptions and knowledge should be those of the user, not those of the computer-sophisticated user-interface designer. For instance, a biochemist studying the geometric structure of molecules is familiar with such concepts as atoms, bonds, dihedral angles, and residues, but does not know and should not have to know such concepts as linked lists, canvases, and event queues.

Learning to use a user interface is similar to learning to use a foreign language. Recall your own foreign-language study. Sentences came slowly, as you struggled with vocabulary and grammar. Later, as practice made the rules more familiar, you were able to concentrate on expanded vocabulary to communicate your thoughts more effectively. The new user of an interactive system must go through a similar learning process. Indeed, if new application concepts must be learned along with new grammar rules and vocabulary, the learning can be even more difficult. The designer's task, then, is to keep the user-interface rules and vocabulary simple, and to use concepts the user already knows or can learn easily.

The language of the user–computer dialogue should be efficient and complete, and should have natural sequencing rules. With an *efficient* language, the user can convey commands to the computer quickly and concisely. A *complete* language allows expression of any idea relevant to the domain of discourse. *Sequencing rules*, which define the order or syntax of the language, should have a minimum number of simple, easy-to-learn cases. Simple sequencing rules help to minimize training and allow the user to concentrate on the problem at hand; complex rules introduce discontinuities and distractions into the user's thought processes.

A user interface may be complete but inefficient; that is, expressing ideas may be difficult and time consuming. For example, a system for logic design needs to provide only a single building block, either the **nor** or the **nand**, but such a system will be laborious to use and thus inefficient. It is better to include in the system a facility for building up more complex commands from the few basic ones.

Extensibility can be exploited to make a language more efficient by defining new terms as combinations of existing terms. Extensibility is commonly provided in operating systems via scripts, cataloged procedures, or command files, and in programming languages via macros, but is less often found in graphics systems.

In person–person dialogue, one person asks a question or makes a statement, and the other responds, usually quite quickly. Even if a reply does not come immediately, the listener usually signals attentiveness via facial expressions or gestures. These are forms of *feedback*, a key component of user–computer dialogue. In both sorts of dialogue, the ultimate response may be provided either in words or with some gesture or facial expression—that is, with a graphic image.

Occasionally, too, the speaker makes a mistake, then says, "Oops, I didn't mean that," and the listener discards the last statement. Being able to undo mistakes is also important in user–computer dialogues.

In a conversation, the speaker might ask the listener for help in expressing a thought, or for further explanation. Or the speaker might announce a temporary digression to another

Samsung/Google, EX1027, p. 408

subject, holding the current subject in abeyance. These same capabilities should also be possible in user–computer dialogues.

With this general framework, let us define the components of the user–computer interface more specifically. Two languages constitute this interface. With one, the user communicates with the computer; with the other, the computer communicates with the user. The first language is expressed via actions applied to various interaction devices, and perhaps also via spoken words. The second language is expressed graphically through lines, points, character strings, filled areas, and colors combined to form displayed images and messages, and perhaps aurally through tones or synthesized words.

Languages have two major components: the meaning of the language, and the form of the language. The *meaning* of a language is its content, or its message, whereas the *form* is how that meaning is conveyed. In person–person dialogue, the meaning "I am happy" can be conveyed with the words "I am happy," or with the words "Ich bin glücklich," or with a smile. In user–computer dialogue, the meaning "delete temp9" might be conveyed by typing the command "DELETE temp9" or by dragging an icon representing file temp9 to a trashcan icon. The form of an interface is commonly called its "look and feel."

There are two elements to meaning in interface design: the conceptual and the functional. There are also two elements to form: sequencing and binding to hardware primitives. The user-interface designer must specify each of these four elements.

The *conceptual design* is the definition of the principal application concepts that must be mastered by the user, and is hence also called the *user's model* of the application. The conceptual design typically defines *objects*, *properties* of objects, *relationships* between objects, and *operations* on objects. In a simple text editor, the objects are characters, lines, and files, a property of a file is its name, files are sequences of lines, lines are sequences of characters, operations on lines are Insert, Delete, Move, and Copy, and the operations on files are Create, Delete, Rename, Print, and Copy. The conceptual design of a user interface is sometimes described by means of a metaphor or analogy to something with which the user is already familiar, such as a typewriter, Rolodex, drafting table and instruments, desktop, or filing cabinet. Although such analogies are often helpful for initial understanding, they can become harmful if they must be stretched unrealistically to explain the more advanced capabilities provided by the computer system [HALA82].

The *functional design* specifies the detailed functionality of the interface: what information is needed for each operation on an object, what errors can occur, how the errors are handled, and what the results of each operation are. Functional design is also called *semantic design*. It defines meanings, but not the sequence of actions or the devices with which the actions are conducted.

The *sequencing design*, part of the form of an interface, defines the ordering of inputs and outputs. Sequencing design is also called *syntactic design*. For input, the sequencing comprises the rules by which indivisible units of meaning (input to the system via interaction techniques) are formed into complete sentences. Units of meaning cannot be further decomposed without loss of meaning. For example, the mouse movements and mouse button clicks needed to make a menu selection do not individually provide information to the application.

Samsung/Google, EX1027, p. 409

For output, the notion of sequence includes spatial and temporal factors. Therefore, output sequencing includes the 2D and 3D layout of a display, as well as any temporal variation in the form of the display. The units of meaning in the output sequence, as in the input sequence, cannot be further decomposed without loss of meaning; for example, a transistor symbol has meaning for a circuit designer, whereas the individual lines making up the symbol do not have meaning. The meanings are often conveyed graphically by symbols and drawings, and can also be conveyed by sequences of characters.

The hardware *binding design*, also called the *lexical design*, is also part of the form of an interface. The binding determines how input and output units of meaning are actually formed from hardware primitives. The input primitives are whatever input devices are available, and the output primitives are the shapes (such as lines and characters) and their attributes (such as color and font) provided by the graphics subroutine package. Thus, for input, hardware binding is the design or selection of interaction techniques, as discussed in Chapter 8. For output, hardware binding design is the combining of display primitives and attributes to form icons and other symbols.

To illustrate these ideas, let us consider a simple furniture-layout program. Its conceptual design has as objects a room and different pieces of furniture. The relation between the objects is that the room contains the furniture. The operations on the furniture objects are Create, Delete, Move, Rotate, and Select; the operations on the room object are Save and Restore. The functional design is the detailed elaboration of the meanings of these relations and operations.

The sequence design might be to select first an object and then an operation on that object. The hardware-binding component of the input language might be to use a mouse to select commands from the menu, to select furniture objects, and to provide locations. The sequence of the output design defines the screen arrangement, including its partitioning into different areas and the exact placement of menus, prompts, and error messages. The hardware-binding level of the output design includes the text font, the line thickness and color, the color of filled regions, and the way in which output primitives are combined to create the furniture symbols.

Section 9.2 discusses some of the fundamental forms a user interface can take; Section 9.3 presents a set of design guidelines that applies to all four design levels. In Section 9.4, we present issues specific to input sequencing and binding; in Section 9.5, we describe visual design rules for output sequencing and binding. Section 9.6 outlines an overall methodology for user-interface design.

9.2 USER-INTERFACE STYLES

Three common styles for user–computer interfaces are *what you see is what you get*, *direct manipulation*, and *iconic*. In this section, we discuss each of these related but distinct ideas, considering their applicability, their advantages and disadvantages, and their relation to one another. There is also a brief discussion of other styles of user–computer interaction: menu selection, command languages, natural-language dialogue, and question–answer dialogue. These are not emphasized, because they are not unique to graphics. (Menus are the closest,

Samsung/Google, EX1027, p. 410

but their use certainly predates graphics. Graphics does, however, permit use of icons rather than of text as menu elements, and provides richer possibilites for text typefaces and fonts and for menu decorations.) None of these styles are mutually exclusive; successful interfaces often meld elements of several styles to meet design objectives not readily met by one style alone.

9.2.1 What You See Is What You Get

What you see is what you get, or *WYSIWYG* (pronounced wiz-ee-wig), is fundamental to interactive graphics. The representation with which the user interacts on the display in a WYSIWYG interface is essentially the same as the image ultimately created by the application. Most, but not all, interactive graphics applications have some WYSIWYG component.

Many text editors (most assuredly a graphics application) have WYSIWYG interfaces. Text that is to be printed in boldface characters is displayed in boldface characters. With a non-WYSIWYG editor, the user sees control codes in the text. For example,

In this sentence, we show @b(bold), @i(italic), and @ub(underlined bold) text.

specifies the following hardcopy output:

In this sentence, we show **bold,** *italic*, and **underlined bold** text.

A non-WYSIWYG specification of a mathematical equation might be something like

@f(@i(u)@sub(max) − @i(u)@sub(min),@i(x)@sub(max) − @i(x)@sub(min))

to create the desired result

$$\frac{u_{max} - u_{min}}{x_{max} - x_{min}}.$$

In such non-WYSIWYG systems, users must translate between their mental image of the desired results and the control codes. Confirmation that the control codes reproduce the mental image is not given until the coded input is processed.

WYSIWYG has some drawbacks. Whenever the spatial and intensity or color resolution of the screen differs from that of the hardcopy device, it is difficult to create an *exact* match between the two. Chapter 13 discusses problems that arise in accurately reproducing color. More important, some applications cannot be implemented with a pure WYSIWYG interface. Consider first text processing, the most common WYSIWYG application. Many text processors provide heading categories to define the visual characteristics of chapter, section, subsection, and other headings. Thus, "heading type" is an object property that must be visually represented. But the heading type is not part of the final hardcopy, and thus, by definition, cannot be part of the display either. There are simple solutions, such as showing heading-type codes in the left margin of the display, but they are counter to the WYSIWYG philosophy. It is for this reason that WYSIWYG is sometimes called "what you see is *all* you get." As a second example, the robot arm in Fig.

Samsung/Google, EX1027, p. 411

7.1 does not reveal the existence of hierarchical relationships between the robot's body, arms, and so on, and it certainly does not show these relationships. These examples are intended not as indictments of WYSIWYG but rather as reminders of its limitations.

9.2.2 Direct Manipulation

A *direct-manipulation user interface* is one in which the objects, attributes, or relations that can be operated on are represented visually; operations are invoked by actions performed on the visual representations, typically using a mouse. That is, commands are not invoked explicitly by such traditional means as menu selection or keyboarding; rather, the command is implicit in the action on the visual representation. This representation may be text, such as the name of an object or property, or a more general graphic image, such as an icon. Later in this section, we discuss the circumstances under which textual and iconic forms of visual representation are appropriate.

The Macintosh interface uses direct manipulation in part, as shown in Fig. 9.1. Disks and files are represented as icons. Dragging a file's icon from one disk to another copies the file from one disk to the other; dragging to the trashcan icon deletes the file. In the earlier Xerox Star, dragging a file to a printer icon printed the file. Shneiderman [SHNE83], who coined the phrase "direct manipulation," discusses other examples of this technique.

Direct manipulation is sometimes presented as being the best user-interface style. It is certainly quite powerful and is especially easy to learn. But the Macintosh interface can be slow for experienced users in that they are forced to use direct manipulation when another

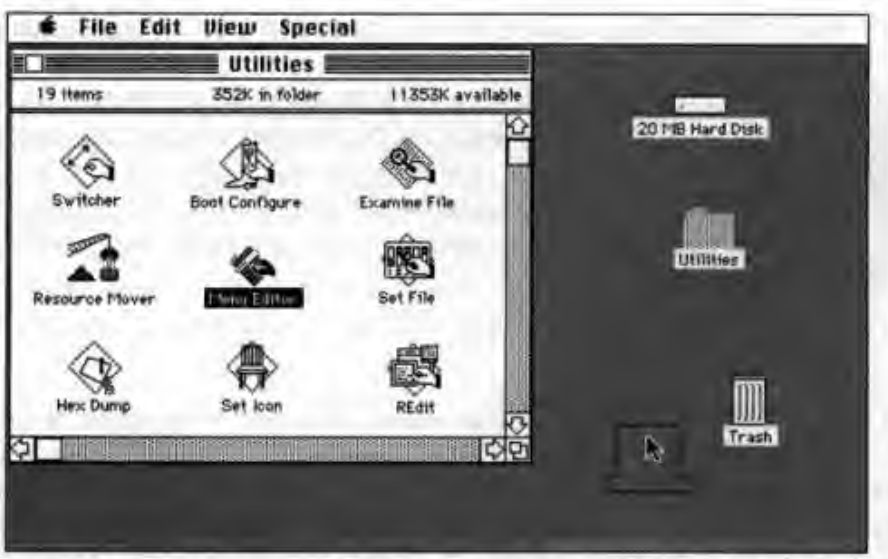


Fig. 9.1 The Macintosh screen. In the upper right is a disk icon; just below it is a directory icon, which is gray-toned to indicate that it is open. At the left is the open directory, with named icons representing the files within it. A file, represented by the icon outline around the cursor, is being dragged to the trashcan at the lower right. (Screen graphics © Apple Computer, Inc.)

Samsung/Google, EX1027, p. 412

style would generally be faster. Printing the file "Chapter 9" with direct manipulation requires the visual representation of the file to be found and selected, then the Print command is involved. Finding the file icon might involve scrolling through a large collection of icons. If the user knows the name of the file, typing "Print Chapter 9" is faster. Similarly, deleting all files of type "txt" requires finding and selecting each such file and dragging it to a trash can. Much faster is the UNIX-style command "rm *.txt", which uses the wild card * to find all files whose names end in ".txt."

An interface combining direct manipulation with command-language facilities can be faster to use than is one depending solely on direct manipulation. Note that direct manipulation encourages the use of longer, more descriptive names, and this tends to offset some of the speed gained from using typed commands. Some applications, such as programming, do not lend themselves to direct manipulation [HUTC86], except for simple introductory flowchart-oriented learning or for those constructs that in specialized cases can be demonstrated by example [MAUL89; MYER86].

Direct-manipulation interfaces typically incorporate other interface styles, usually commands invoked with menus or the keyboard. For instance, in most drafting programs, the user rotates an object with a command, not simply by pointing at it, grabbing a handle (as in Section 8.3.3), and rotating the handle. Indeed, it is often difficult to construct an interface in which all commands have direct-manipulation actions. This reinforces the point that a single interaction style may not be sufficient for a user interface: Mixing several styles is often better than is adhering slavishly to one style.

The form fill-in user interface is another type of direct manipulation. Here a form is filled in by pointing at a field and then typing, or by selecting from a list (a selection set) one of several possible values for the field. The limited functional domain of form fill-in and its obvious correspondence to filling in real forms makes direct manipulation a natural choice.

WYSIWYG and direct manipulation are separate and distinct concepts. For instance, the textual representation of a graphics image can be modified via direct manipulation, and the graphical image of a WYSIWYG system can be modified purely by a command-language interface. Especially when used together, however, the two concepts are powerful, easy to learn, and reasonably fast to use, as many successful user interfaces have demonstrated.

9.2.3 Iconic User Interfaces

An *icon* is a pictorial representation of an object, an action, a property, or some other concept. The user-interface designer often has the choice of using icons or words to represent such concepts. Note that the use of icons is not related to the direct-manipulation issue: Text can be directly manipulated just as well as icons can, and text can represent concepts, sometimes better than icons can.

Which is better, text or icons? As with most user-interface design questions, the answer is, "it depends." Icons have many advantages. Well-designed icons can be recognized more quickly than can words, and may also take less screen space. If carefully designed, icons can be language-independent, allowing an interface to be used in different countries.

Samsung/Google, EX1027, p. 413



Fig. 9.2 Icons used to represent common office objects.

Icon design has at least three separate goals, whose importance depends on the specific application at hand:

1. *Recognition*—how quickly and accurately the meaning of the icon can be recognized
2. *Remembering*—how well the icon's meaning can be remembered once learned
3. *Discrimination*—how well one icon can be distinguished from another.

See [BEWL83] for a report on experiments with several alternative icon designs; see [HEME82; MARC84] for further discussion of icon-design issues.

Icons that represent objects can be designed relatively easily; Fig. 9.2 shows a collection of such icons from various programs. Properties of objects can also be represented easily if each of their values can be given an appropriate visual representation. This certainly can be done for the properties used in interactive graphics editors, such as line thickness, texture, and font. Numeric values can be represented with a gauge or dial icon, as in Fig. 8.21.

Actions on objects (that is, commands) can also be represented by icons. There are several design strategies for doing this. First, the command icon can represent the *object* used in the real world to perform the action. Thus, scissors can be used for Cut, a brush for Paste, and a magnifying glass for Zoom. Figure 9.3 shows a collection of such command icons. These icons are potentially difficult to learn, since the user must first recognize what the icon *is*, then understand what the object represented *does*. This two-step understanding process is inherently less desirable than is the one-step process of merely recognizing what object an icon represents. To complicate matters further, suppose that the object might be used for several different actions. A brush, for example, can be used for spreading paste (to

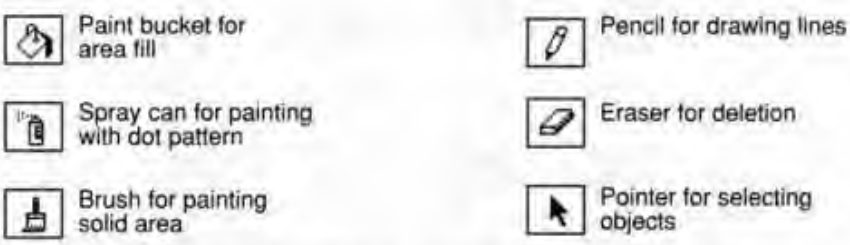


Fig. 9.3 Command icons representing objects used to perform the corresponding command. (Copyright 1988 Claris Corporation. All rights reserved.)

Samsung/Google, EX1027, p. 414

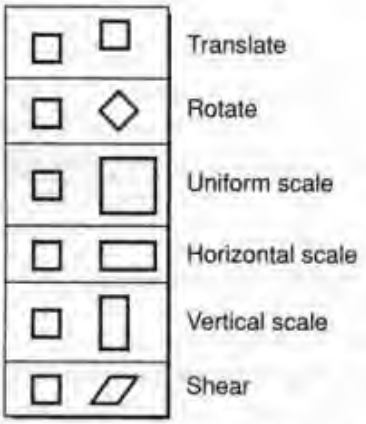


Fig. 9.4 Command icons indicating geometric transformations by showing a square before and after the commands are applied.

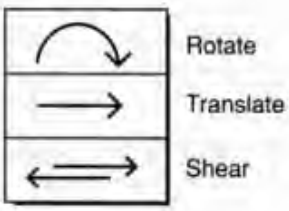


Fig. 9.5 Several abstract command icons for some of the actions depicted in Fig. 9.4. Not all geometric operations can be represented in this way.

paste something in place), and also for spreading paint (to color something). If both Paste and Paint could reasonably be commands in the same application, the brush icon could be ambiguous. Of course, sometimes only one interpretation will make sense for a given application.

Another design strategy for command icons is to show the command's *before and after* effects, as in Fig. 9.4 and Color Plates I.19–I.21. This works well if the representations for the object (or objects) are compact. If the command can operate on many different types of objects, however, then the specific object represented in the icon can mislead the user into thinking that the command is less general than it really is.

The NeXT user interface, implemented on a two-bit-per-pixel display, uses icons for a variety of purposes, as seen in Color Plate I.22.

A final design approach is to find a more *abstract representation* for the action. Typical examples are shown in Fig. 9.5. These representations can depend on some cultural-specific concept, such as the octagonal stop-sign silhouette, or can be more generic, such as X for Delete.

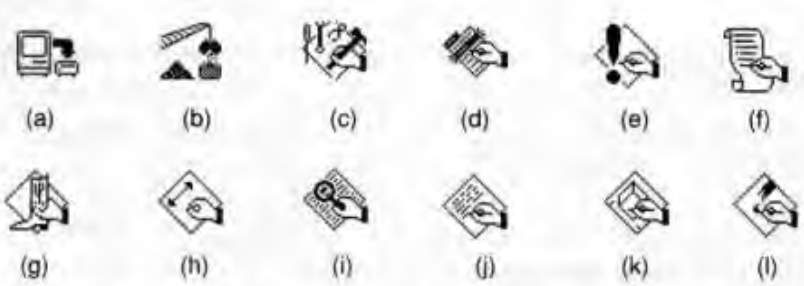


Fig. 9.6 Icons that represent Macintosh programs. What does each icon represent? In most cases, the icons suggest the type of information that is operated on or created. See Exercise 9.14 for the answers.

Samsung/Google, EX1027, p. 415

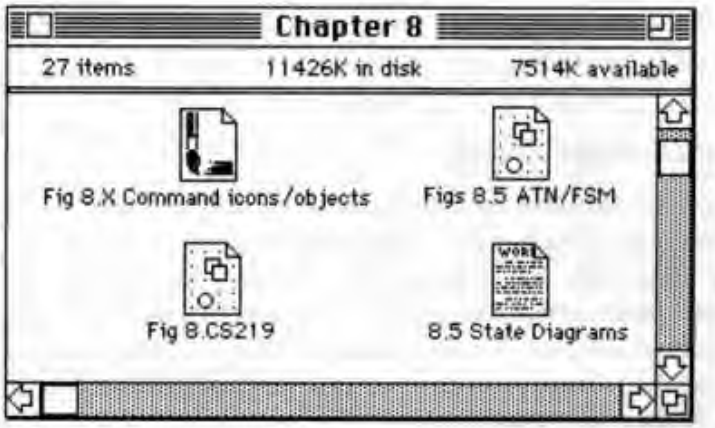


Fig. 9.7 The contents of a disk directory represented with icons and text. The icons help to distinguish one file from another. (Certain screen graphics © Apple Computer, Inc.)

Arbitrarily designed icons are not necessarily especially recognizable. Figure 9.6 shows a large number of icons used to represent Macintosh programs. We challenge you to guess what each program does! However, once learned, these icons seem to function reasonably well for remembering and discrimination.

Many visual interfaces to operating systems use icons to discriminate among files used by different application programs. All files created by an application share the same icon. If a directory or disk contains many different types of files, then the discrimination allowed by the icon shapes is useful (see Fig. 9.7). If all the files are of the same type, however, this discrimination is of no use whatsoever (see Fig. 9.8).

Icons can be poorly used. Some users dislike icons such as the trashcan, contending that such ideas are juvenile, "cute," and beneath their dignity. The designer may or may not agree with such an evaluation, but the user's opinion is usually more important than is

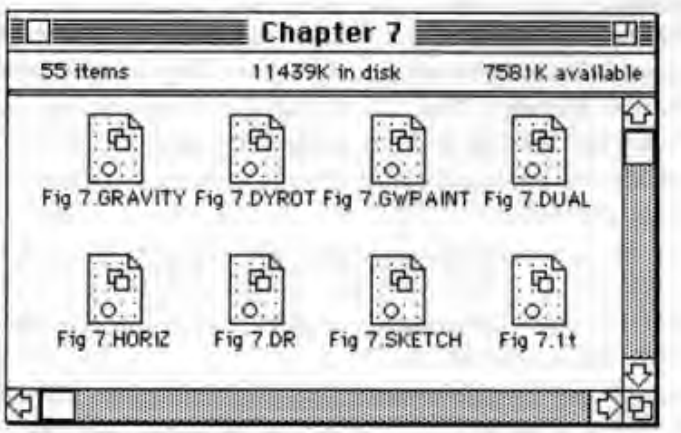


Fig. 9.8 The contents of a disk directory represented with icons and text. Since the files are all of the same type, the icons do not help to distinguish one file from another, and simply take up extra space. (Computer screen graphics © Apple Computer, Inc.)

Samsung/Google, EX1027, p. 416

the designer's. The user who dislikes a computer or program and thus develops a negative attitude is best taken seriously.

9.2.4 Other Dialogue Forms

The dialogue styles discussed in the previous section might be called "intrinsically graphical" in that our focus has been on graphically oriented interaction. A number of other dialogue forms are not intrinsically graphical but can be used in graphical applications. Four such forms are menus, command languages, natural-language dialogue, and question–answer dialogue. We have discussed many specific design issues concerning menus in the previous chapter. In this section, we briefly discuss more general issues involving each of these dialogue forms.

Menus are widely used in both graphical and nongraphical applications. In either case, however, the fundamental advantage of menus is that the user can work with what is called *recognition memory*, where visual images (textual or iconic menu items) are associated with already-familiar words and meanings. This contrasts with *recall memory*, where the user must recall from memory a command or concept in order to enter information into the computer. Menus reduce the memory load for users, and hence are especially attractive to novices. Menus, along with form fill-in, allow current selections to be indicated visually, further reducing the user's memory load and also allowing rapid input if the current selection is desired. On the other hand, menus limit the size of the selection set of alternatives, whereas some of the other dialogue styles do not.

Use of a *command language* is the traditional way to interact with a computer. This technique can accommodate large selection sets, is easy to extend (just add another command), and is fairly fast for experienced users who can type. Learning time is its major liability, with the need for typing skills a second factor. Errors are more likely with command languages than with menus, because of the possibility of typing and recall errors.

Natural-language dialogue is often proposed as the ultimate objective for interactive systems: If computers could understand our commands, typed or spoken in everyday English, then everyone would be able to use them. However, current voice recognizers with large vocabularies must be individually trained to recognize a particular user's voice; they also make mistakes, and must be corrected somehow. Typing long sentences is tedious. Also, because natural language does not bound the command set that an application program must handle, and also can be quite ambiguous, users of natural-language interfaces tend to make requests that cannot be fulfilled, which leads to frustration of the user and poor performance of the system.

This problem can be overcome in limited-domain (and hence limited-vocabulary) natural-language systems, in which users are familiar with the system's capabilities and hence are unlikely to make unreasonable requests. Drawing programs and operating systems are examples of such systems.

There is a fundamental flaw, however, in the argument that natural language interaction is the ultimate objective. If the argument were true, we would be satisfied to interact with one another solely by means of telephone and/or keyboard communications. It is for this reason that voice input of natural language to an interactive graphics application program is

Samsung/Google, EX1027, p. 417

TABLE 9.1 COMPARISON OF SEVEN USER INTERFACE STYLES

	WYSI-WYG*	Direct manipulation	Menu selection	Form fill-in	Command language	Natural language	Q/A dialogue
learning time	low	low	med	low	high	low	low
speed of use		med	med	high	high	med	low
error-proneness	low	low	low	low	high	high	low
extensibility	low	low	med	med	high	high	high
typing skill required		none	none	high	high	high**	high

*WYSIWYG has several blank fields because it is not a complete interface style, since it must be accompanied by some means of entering commands.
**Assuming keyboard input; none for voice-recognizer input.

most likely to be used in combination with other dialogue styles, to allow overlapped use of the voice and hands to speed interaction. After all, this is exactly how we work in the real world: we point at things and talk about them. This powerful concept was compellingly demonstrated a decade ago in the "Put-that-There" [BOLT80; BOLT84] program for manipulating objects. In this system, the user can move an object by pointing at it while saying "put that;" pointing elsewhere, and saying "there." A recent study of a VLSI design program using voice input of commands combined with mouse selection of objects and positions found that users worked 60 percent faster than did those who had just a mouse and keyboard [MART89].

Question–answer dialogue is computer-initiated, and the user response is constrained to a set of expected answers. Using a keyboard for input, the user can give any answer. If the set of expected answers is small, the question can include the possible answers; menu selection might even be provided instead of typing as a means for the user to respond. In the limit, question–answer dialogue becomes a sequential set of menu selections. A common failing of instances of this dialogue form is the inability to go back several steps to correct an answer. A more general problem with the sequentiality implied by this form is that of context: The user has only the context of the past and current questions to assist in interpreting the current question. With a form fill-in dialogue, by contrast, the user can see all the fields to be entered, and so can quickly tell, for instance, whether an apartment number in an address goes in the street-address field or in a separate apartment-number field.

Table 9.1 compares user-interface dialogue styles. A much more extensive discussion of the pros and cons of many of these styles can be found in [SHNE86].

9.3 IMPORTANT DESIGN CONSIDERATIONS

In this section, we describe a number of design principles to help ensure good human factors in a design: be consistent, provide feedback, minimize error possibilities, provide error recovery, accommodate multiple skill levels, and minimize memorization. Application of these principles is generally considered necessary, although by no means is sufficient, for a successful design. These and other principles are discussed more fully in [FOLE74; GAIN84; HANS71; MAYH90; RUBE84; SHNE86].

Samsung/Google, EX1027, p. 418

9.3.1 Be Consistent

A *consistent* system is one in which the conceptual model, functionality, sequencing, and hardware bindings are uniform and follow a few simple rules, and hence lack exceptions and special conditions. The basic purpose of consistency is to allow the user to *generalize* knowledge about one aspect of the system to other aspects. Consistency also helps to avoid the frustration induced when a system does not behave in an understandable and logical way. The best way to achieve this consistency is through a careful top-down design of the overall system.

Simple examples of consistency in the output portion of a user interface are

- The same codings are always employed. Colors always code information in the same way, just as red always means stop, and green always means go.
- System-status messages are shown at a logically (although not necessarily physically) fixed place.
- Menu items are always displayed in the same relative position within a menu, so that users can allow "muscle memory" to help in picking the desired item.

Examples of consistency in the input portion are

- Keyboard characters—such as carriage return, tab, line feed, and backspace—always have the same function and can be used whenever text is being input.
- Global commands—such as Help, Status, and Cancel—can be invoked at any time.
- Generic commands—such as Move, Copy, and Delete—are provided and can be applied, with predictable results, to any type of object in the system.

We should, however, remember Emerson's observation that "A foolish consistency is the hobgoblin of little minds" [EMER03]. Consistency can conflict with other design objectives. For instance, if dragging a file icon to the trash deletes the file, what should happen when a file icon is dragged to an electronic-mail outbox? Should the file be sent and then deleted, for consistency with dragging to the trash? Or should a copy of the file be sent? If a file is dragged to a printer icon to print the file, should the file be printed and then deleted, for consistency with dragging to the trash? Surely in these latter two cases the file should not be deleted. The *law of least astonishment*, a higher design principle, suggests that doing what the user is likely to consider normal or reasonable is more important than is maintaining pure consistency.

Figure 9.9 shows how state diagrams can help to identify inconsistency. We can see here that help is available only from the move state, not from the other states. A mixed strategy is used to let users change their minds once an action sequence has been initiated. From the move and delete states there is a Cancel command, whereas the rotate state has an Undo command. The sequence of object/operation varies: for Move and Delete, the sequence is operation then object, whereas it is the reverse for rotation. The feedback strategy is also mixed: dynamic for moving, static for rotation.

Reisner demonstrated experimentally an intuitively expected result: Given two functionally equivalent user interfaces, new users make fewer errors and learn more quickly with

Samsung/Google, EX1027, p. 419

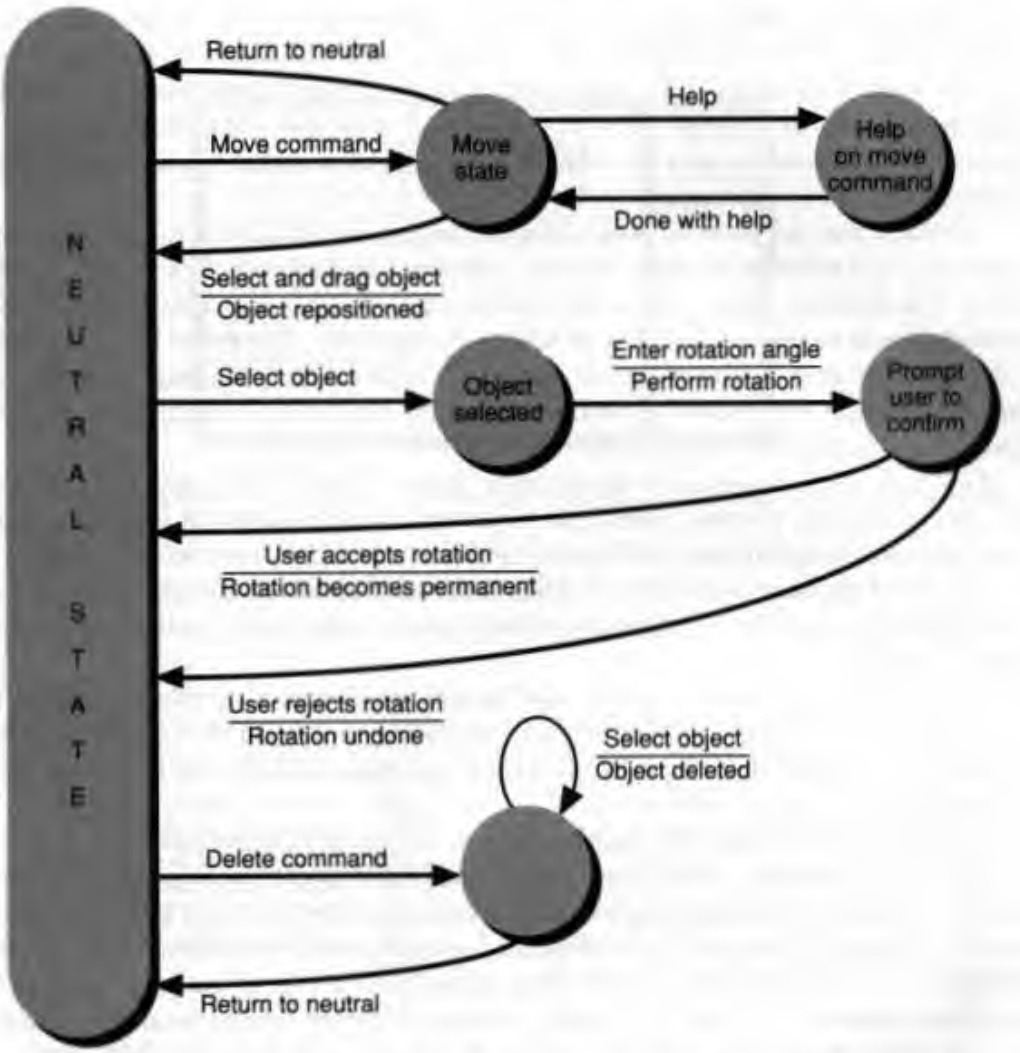


Fig. 9.9 State diagram of a user interface with an inconsistent syntax.

one that has a simpler syntactic structure [REIS82]. Thus, another useful design objective to apply is simply to minimize the number of different syntactic structures.

At the functional level, consistency requires the use of *generic commands* that apply across as broad a range as possible. For example, chairs and desks should be moved in the same way in the room-layout program discussed previously; files should be opened, deleted, and saved from within application programs with the same generic file-manipulation commands.

9.3.2 Provide Feedback

Have you ever tried conversing with a partner who neither smiles nor nods, and who responds only when forced to do so? It is a frustrating experience, because there is little or no indication that the partner understands what you are saying. Feedback is as essential in

Samsung/Google, EX1027, p. 420

conversation with a computer as it is in human conversation. The difference is that, in normal conversation with another person, many sources of feedback (gestures, body language, facial expressions, eye contact) are usually provided without conscious action by either participant. By contrast, a workstation gives little automatic feedback (just the "power on" light and perhaps the whir of a fan), so all feedback must be planned and programmed.

Feedback can be given at three possible levels, corresponding to the functional, sequencing, and hardware-binding (semantic, syntactic, and lexical) levels of user-interface design. The designer must consciously consider each level and explicitly decide whether feedback should be present, and, if it should be, in what form. The lowest level of feedback is the hardware level. Each user action with an input device should cause immediate and obvious feedback: for instance, characters typed on a keyboard are echoed, and mouse movements are echoed by the cursor.

Feedback at the sequencing-design level occurs as each unit (word) of the input language (command, position, object, etc.) is accepted by the system. A selected object or menu command is highlighted, so the user can know that actions have been accepted (i.e., the "words" have been understood). Similar forms of feedback are prompting for the next input, lighting the function key that has just been pressed, and echoing verbal (speech) input with text output.

Another form of sequencing feedback should occur when a complete sequence has been input and is found to be well formed. This acknowledgment of receipt of a proper command is generally needed only if processing the command itself will take more than 1 or 2 seconds.

Another type of functional feedback—some indication that the computer is at least working on the problem—is necessary only if completion of the operation will take more than a few seconds. (In the absence of such feedback, users have been known to express their frustration physically on the workstation, or even on the application designer!) Such feedback can take many forms; particularly attractive is a dial or gauge to indicate the percentage complete. The user can quickly determine whether a coffee break is in order. In an experiment, Myers found a strong user preference for such indicators [MYER85].

The most useful and welcome form of functional feedback tells the user that the requested operation has been completed. This is usually done with a new or modified display that explicitly shows the results of the operation.

It is useful to distinguish between problem-domain and control-domain feedback. *Problem-domain* feedback concerns the actual objects being manipulated: their appearance, their position, their existence. *Control-domain* feedback has to do with the mechanisms for controlling the interactive system: status, current and default values, menus, and dialogue boxes.

Problem-domain feedback is needed if users can see just part of a large drawing, so that they can know which part of the world is being displayed. Figure 9.10 shows one way to do this. The approach can be even more effective with two displays—one for the overview, the other for the detail. In either case, the rectangle in the overview indicates which part of the drawing is being shown in the detailed display. Panning and zooming are generally effected by dragging and resizing the overview rectangle. Figure 9.11 shows how increasing

Samsung/Google, EX1027, p. 421

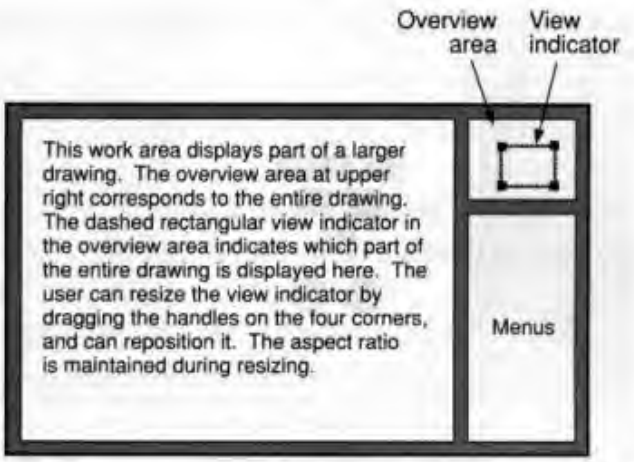


Fig. 9.10 The view indicator is used both to give an overview and to control what is displayed in the work area.

(a) Scroll bar with no status display to indicate which part of the drawing is being displayed, or to indicate the size of the window with respect to the overall drawing.



(b) Scroll bar with "elevator" to indicate which part of the drawing is being displayed. There is still no way for the user to know the size of the window with respect to the overall drawing.



(c) A page number has been added to the elevator, so there is now partial information to help the user infer the size of the window with respect to the overall drawing.



(d) The elevator has now been scaled so that the ratio of its height to the height of the scroll bar is the same as the ratio of the window's height to that of the overall drawing. The page number is now a form of redundant coding. It conveys linguistically some of the information that the elevator conveys graphically.



Fig. 9.11 Four different levels of feedback in scroll bars, ranging from none in (a) to redundant coding in (d).

Samsung/Google, EX1027, p. 422

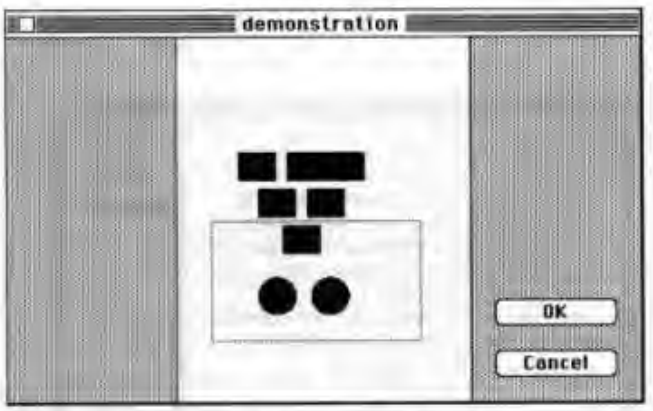


Fig. 9.12 The overview mode in MacPaint. Because the screen is very small, the overview alternates with the normal, more detailed view of the drawing. The user drags the rectangular dashed box to the desired area, and selects "OK" to see the detailed view of the enclosed area. (Copyright 1988 Claris Corporation. All rights reserved.)

amounts of feedback can be built into window scroll bars. Another approach to orienting the viewer is used in MacPaint, as shown in Fig. 9.12.

An important type of control-domain feedback is *current settings*. Current settings can be shown in a feedback area. If the menus or other tools by which settings are selected can always be displayed, then the current setting can be indicated there, as in Fig. 9.13. The pull-out menus illustrated in Fig. 8.14 also show current settings.

The positioning of feedback is important. There is a natural tendency to designate a fixed area of the screen for feedback and error messages. This can destroy *visual continuity*, however, since the user's eyes must move between the work area and the message area. Indeed, users often do not notice messages in these fixed areas. Adding audio feedback can eliminate this problem.

Placing the feedback where the user is looking, which is generally at or near the cursor, is especially attractive. Tilbrook's Newswhole system [TILB76] was one of the first to employ this idea; it uses a seated Buddha to encourage the user to be patient during computational delays, and a thumbs-down symbol to indicate a mistake.

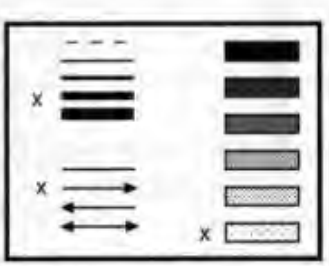


Fig. 9.13 In this menu of graphic attributes, the x indicates the current setting. The user has constant feedback if the menu can be permanently visible. If not, a more compact form of permanent feedback should be used.

Samsung/Google, EX1027, p. 423

9.3.3 Minimize Error Possibilities

Don't set the user up for a fall is another way to state this objective. For example,

- Do not offer menu options that will elicit an "illegal selection, command not valid now" message
- Do not let the user select Delete if there is nothing to be deleted
- Do not let the user try to change the font of the currently selected object if the object is not a text string
- Do not let the user Paste when the clipboard is empty
- Do not let the user Copy when nothing is selected to be copied
- Do not let the user select a curve-smoothing command when the currently selected object is not a curve.

In all these instances, the system should instead disable unavailable items and alert the user by changing the menu item's appearance—for instance, by making it gray instead of black.

These are all examples of *context sensitivity*: The system provides the user with only those commands that are valid in the current context or mode. When there is a context, the system should guide the user to work within that context and should make it difficult or impossible for the user to do things that are not permissible in that context.

Another aspect of this objective is to avoid *side effects*, which are results the user has not been led to expect. The classic side effect is the print command that also deletes the file being printed. Side effects arise from poor design or ineffective communication with the user regarding what a command does.

9.3.4 Provide Error Recovery

We all make mistakes. Imagine not having a backspace key on your computer keyboard! The effect on your productivity, as you became much more cautious in typing, would be devastating. There is ample experimental evidence that people are more productive if their mistakes can be readily corrected. With good error recovery, the user is free to explore unlearned system facilities without "fear of failure." This freedom encourages exploratory learning, one of the major ways in which system features are learned. We discuss four types of error recovery: Undo, Abort, Cancel, and Correct.

The most serious type of error is functional: the user has mistakenly invoked one or a series of commands and has obtained unanticipated results. An *Undo* command is needed to reverse the results of the command. There are two types of undo: single level and multilevel. The single-level undo can reverse only the most recently executed command. This Undo itself is a command, so the second of two successive Undo commands undoes the first Undo, returning the system to its state prior to the two Undo commands.

In contrast, a multilevel Undo operates on the stack of previous commands. The actual number of commands stacked up and thus able to be undone is implementation-dependent; in some cases, all commands since the session began are saved. With a multilevel undo, a *Redo* command is also needed, so that, if the user backs up too far in the command stack, the most recently undone command can be redone. Neither Undo nor Redo is entered on

Samsung/Google, EX1027, p. 424

the stack. Several tricky issues concerning Redo are discussed in [VITT84]. For a single level, Undo and Redo are mutually exclusive: One or the other, but not both, can be available at any given time. For multilevel, Undo and Redo can be available at the same time.

Users often need help in understanding the scope of the Undo command (i.e., how much work the command will undo). They also are often confused about whether Undo applies to windowing commands, such as scrolling, as well as to application commands. If the Undo command is in a menu, the menu-item text can indicate what will be undone; instead of "undo," the entry could be "undo copy" or "undo deletion." Kurlander and Feiner have developed a graphical way to show the history of user actions and the scope of Undo, shown in Color Plate I.23 [KURL88; KURL90].

A form of undo is sometimes provided as an *explicit-accept, explicit-reject* step. After a command is carried out and its results are shown on the display, the user must accept or reject the results before doing anything else. This step adds to the number of user actions required to accomplish a task, but does force the user to think twice before confirming acceptance of dangerous actions. However, an Undo command is usually preferable because an explicit action is required only to reject the results of a command; the command is implicitly accepted when the next command (other than the Undo command) is entered. Hence, we call undo an *implicit-accept, explicit-reject* strategy.

No matter how undo is provided, its implemention requires extra programming, especially for commands that involve major changes to data structures. An easier, although less satisfactory, alternative to undo is to require the user explicitly to confirm commands for which there is no undo. This is commonly used for the file-delete command.

A user may realize that a functional-level mistake has been made while a command is still being performed. This illustrates the need for an *Abort* command to terminate prematurely a currently executing command. Like Undo, Abort must restore the system to its exact state prior to initiation of the aborted command. In fact, Abort and Undo can be thought of as essentially the same command: They both reverse the most recently specified functional-level action. A user-interface design might make both actions available with the same name.

A less dramatic type of error occurs when the user is partway through specifying information required to carry out some command, and says, "Oops, I don't really want to do this!" A poorly designed interface gives the user no choice but to proceed with the command, after which an Undo or Abort (if available) is used to recover. A well-designed interface lets the user back out of such a situation with a *Cancel* command. This is especially common with a form fill-in dialogue, where a Cancel command is often available on the form, as in Fig. 8.32. Note that Cancel can also be thought of as a specialized Undo command, with the system reverting to the state prior to the current command.

In a less serious type of error, the user may want to *correct* one of the units of information needed for a command. The dialogue style in use determines how easy to make such corrections are. Command-language input can be corrected by multiple backspaces to the item in error, followed by reentry of the corrected information and all the information that was deleted. If the system has line-editing capabilities, then the cursor can be moved back to the erroneous information without the intervening information being deleted. Form fill-in allows simple corrections as well, whereas question–answer and menu dialogues are not so forgiving. The dynamic interaction techniques discussed in Chapter 8 provide a form

Samsung/Google, EX1027, p. 425

of error recovery: for instance, the position of an object being dragged into place is easy to change.

9.3.5 Accommodate Multiple Skill Levels

Many interactive graphics systems must be designed for a spectrum of users, ranging from the completely new and inexperienced user through the user who has worked with the system for thousands of hours. Methods of making a system usable at all skill levels are accelerators, prompts, help, extensibility, and hiding complexity.

New users normally are most comfortable with menus, forms, and other dialogue styles that provide considerable prompting, because this prompting tells them what to do and facilitates learning. More experienced users, however, place more value on speed of use, which requires use of function keys and keyboard commands. Fast interaction techniques that replace slower ones are called *accelerators*. Typical accelerators, such as one-letter commands to supplement mouse-based menu selection, have been illustrated in previous sections. The Sapphire window manager [MYER84], taking this idea even further, provides three rather than two ways to invoke some commands: pointing at different areas of the window banner and clicking different mouse buttons, a standard pop-up menu, and keyboard commands.

The Macintosh uses accelerators for some menu commands, as was shown in Fig. 8.13. Another approach is to number menu commands, so that a number can be typed from the keyboard, or a command can be selected with the cursor. Alternatively, the command name or abbreviation could be typed.

One of the fastest accelerators is the use of multiple clicks on a mouse button. For instance, the Macintosh user can select a file (represented as an icon) by clicking the mouse button with the cursor on the icon. Opening the file, the typical next step, can be done with a menu selection, an accelerator key, or an immediate second button click. The two rapid clicks are considerably faster than is either of the other two methods. From within applications, another scheme is used to open files, as illustrated in Fig. 9.14. The dialogue box permits a file name to be selected either by pointing or by typing. If the name is typed,

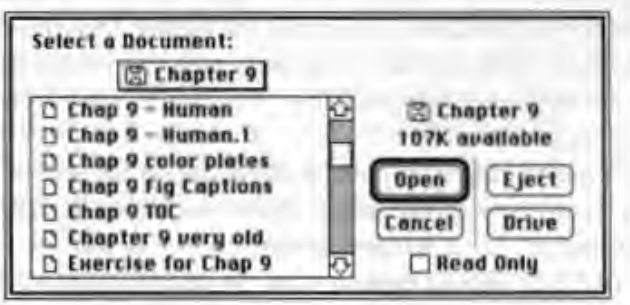


Fig. 9.14 Opening files from within a Macintosh program. The user enters the Open command, either by menu selection or with a two-key chord, causing the dialogue box to appear. The highlighted file can be opened with the "open" button or with the carriage-return key. The user can highlight a new file by selecting it with the cursor or by typing some or all of its name. Therefore, the user can open a file using only the keyboard, by entering the two-key chord, a partial file name, and the return key. (Computer screen graphics © Apple Computer, Inc.)

Samsung/Google, EX1027, p. 426

autocompletion permits the user to type only enough characters for the name to be specified unambiguously. Double clicking on a file name opens the file immediately.

Another form of accelerator is to provide command-line input as an alternative to the other styles. As users gain experience, they use the command line more and more. This transition can be aided by displaying the command-line equivalent of commands that are entered in other ways.

Unlike feedback, which acknowledges specific user actions, the purpose of *prompts* is to suggest what to do next. The more experienced the user, the less prompting is appropriate, especially if prompting is obtrusive and slows down the interaction or uses much of the screen. Many systems provide several levels of prompting controllable by the user; the inexperienced can be "led by the hand," whereas the experienced can proceed without the prompts.

Prompting can take many forms. The most direct is a displayed message that explains explicitly what to do next, such as "Specify location." A speech synthesizer can give explicit aural instructions to the user. Subtler forms of prompting are also available. On a function-key box, buttons eligible for selection can be illuminated. A prominent tracking cross or cursor can be displayed when a position must be input; a blinking underline cursor can indicate that a text string is to be input; a scale or dial can be displayed when a value is desired. Direct-manipulation graphical interfaces implicitly provide prompts: the icons that can be manipulated are the prompt.

A *help* facility allows the user to obtain additional information about system concepts, typical tasks, various commands and the methods used to invoke them, and interaction techniques. Ideally, help can be requested from any point in the interactive dialogue, always with the same mechanism. The return from help should leave the system in exactly the same state as when help was invoked, and the help should be context-sensitive. For example, if help is invoked while the system is awaiting a command, a list of commands available in this state should be shown (with menus or function keys, this may be unnecessary). The Help command followed by a command name should yield more information about the command. If help is requested while the parameters of a command are being entered, details about the parameters should be provided. A second Help command should produce more detailed information and perhaps allow more general browsing through online documentation. Sukaviriya [SUKA88] developed a system to show the user an animation of how to accomplish a task, using the current context as the basis of the animation. Some help capabilities based on hypertext systems allow the user to follow complex sets of links among various help topics.

An easy way to invoke help is to point at the entity on the screen about which help is desired. The entity could be a menu item, a status indicator (the help should explain the status and how to change it), a window banner, a scroll bar, or a previously created application object (the help should explain what the object is and what operations can be applied to it). This approach, however, can be used only for visible objects, not for more abstract concepts or for tasks that must be performed with a series of commands.

A help capability is appropriate even if prompts and menus are displayed, because it gives the user an opportunity to receive more detailed information than can be provided in a short prompt. Even experienced users forget details, particularly in a large and complex application.

Samsung/Google, EX1027, p. 427

Expert systems are beginning to be integrated into user interfaces to provide help that not only is context-sensitive, but also is tailored to individual user profiles. These profiles are developed automatically by the system as a new user and the system interact and learn more about each other, just as teachers learn about their students and custom-tailor their suggestions.

Making the user interface *extensible* means letting the user add additional functionality to the interface by defining new commands as combinations of existing commands. The key is to be able to save and replay sequences of user actions. A particularly appealing macro-definition capability is one in which user actions are automatically saved in a trace file. To create a macro, the user edits the trace file to identify the start and end of the macro, replace literals with parameters, and names the macro. Several commercial applications, such as Ashton-Tate's Full Impact, have such capabilities; Olsen has developed a particularly sophisticated prototype system [OLSE88].

Hiding complexity can allow new users to learn basic commands and to start doing productive work without becoming bogged down with specifying options, learning infrequently used specialized commands, or going through complicated start-up procedures. On the other hand, powerful systems of necessity have many commands, often with many variations. The solution to this quandary is to design the entire set of commands so that it has a small "starter kit" of commands. Default values (current or initial settings) that follow the law of least astonishment can often be useful to achieve this goal.

For example, a chart-making program should allow the user to request a pie chart, specify some data, and immediately see the chart. If the user is dissatisfied with some details of the chart layout, she should be able, say, to modify the radius of the pie, to change the color or texture of each pie slice, to add annotations, to change the text face or font used to display the data values, or to change the position of the data values displayed with each pie slice. But the user should not be forced to specify each of these explicitly when initially creating the chart.

Another design strategy is to make complicated and advanced commands available only via keyboard commands or function keys. This approach keeps the menus smaller and makes the system simpler and less intimidating. Alternatively, two or more sets of menus can be provided, each with successively more commands included.

Yet another way to hide complexity is to use control keys to modify the meaning of other commands. For instance, the Macintosh window manager normally activates the window that the user selects and drags to a new position. The more advanced user can reposition a window without activating it by holding down the control key (called the *command key* on the Macintosh, in appropriate deference to computer-naive users). New users simply are not told about this feature.

9.3.6 Minimize Memorization

Interface designs sometimes force unnecessary memorization. In one design-drafting system, objects are referred to by numeric rather than by alphanumeric names. To appreciate what that means, we can imagine an interactive operating system in which file names are numeric. The remembering and learning tool of mnemonic names would be unavailable, forcing rote memorization or the use of auxiliary written aids. Of course,

Samsung/Google, EX1027, p. 428

explicit picking of displayed objects or icons further eliminates the need for memorization. It is important to invoke the user's recognition rather than recall memory whenever possible.

In one interactive graphing system, a command such as "Plot years gross net" produces a trend chart of yearly gross income and net income on a single set of axes. A reasonable way to control the style of a line is to use a command such as "Linestyle net dash" (to plot the net income with a dashed line). Unhappily, the actual command is of the form "Linestyle 3 dash." The "3" refers to the third variable named in the most recent "Plot" command—in this case, net. Since the most recent Plot command is not generally on the screen, the user must remember the order of the parameters.

Some help systems completely obscure the work area, forcing the user to memorize the context in order to interpret the help message. Then, once he understands the help information, the user must remember it while returning to the context in which the error occurred. Window managers solve this problem; help information is in one window, the application is in another.

9.4 MODES AND SYNTAX

Loosely defined, a *mode* is a state or collection of states in which just a subset of all possible user-interaction tasks can be performed. Examples of modes are these:

- A state in which only commands applicable to the currently selected object are available
- A state in which a dialogue box must be completed before another operation can be performed
- A state for making drawings in a document-preparation system in which separate programs are used to edit text, to make drawings, and to lay out the document
- A state in which available commands are determined by the current data-tablet overlay.

Thus, modes provide a context within which the system and user operate.

There are two kinds of modes: harmful ones and useful ones. A *harmful mode*, as discussed by Tesler [TESL81] and by Smith and colleagues [SMIT82], lasts for a period of time, is not associated with any particular object, is not visible to the user, and serves no meaningful role. Harmful modes confuse users; users get stuck in them and cannot get out, or users forget in which mode they are and attempt to invoke commands that are not available, potentially creating errors. Modes that decrease user productivity are harmful.

On the other hand, *useful modes* narrow the choices of what to do next, so prompts and help can be more specific and menus can be shorter and thus easier to traverse. A well-organized mode structure can reduce the burden on the user's memory and can help to organize knowledge about the interface into categories based on the mode. Useful modes increase user productivity.

Useful modes clearly indicate the current mode, provide feedback to show what commands are available, and include an easy, obvious, and fast means for exiting from the mode. Window managers provide highly visible modes, since each window represents a different mode; mode switching is effected by deactivating one window and activating another.

Samsung/Google, EX1027, p. 429

Users can be made aware of short-lived modes by a state of heightened muscle tension while in the mode. The "button-down–dynamic feedback–button-up" interaction techniques discussed in Chapter 8 make the user aware of the mode through the muscle tension involved in holding down the mouse button [BUXT86].

Command-language syntax has a major influence on the mode structure of a user interface. The traditional prefix syntax of Command, parameter 1, . . ., parameter *n* locks the user into a mode as soon as the command is specified: Only parameters can be entered, possibly in a required order. Mechanisms for error correction (Section 9.3.4) are especially important here, because otherwise the user who erroneously selects a command must continue the potentially lengthy parameter-specification process.

One of the difficulties of prefix syntax is solved by a process called *factoring* or *orthogonalization*. Commands are provided for setting each of the parameters to a current value; parameters may also have an initial (default) value. Consider, for instance, the following unfactored command syntax, where an initial capital letter indicates a command, and lowercase letters indicate parameters:

```
Draw_line        point point line_style line_thickness line_intensity
```

We can factor out the attribute specifications into either three separate commands or one overall attribute-setting command. Hence, the user would go through the sequence

```
Set_attributes   attribute_values   {Only if the current attribute values
                                     are inappropriate}
Draw_line        point point
```

We can factor this sequence further by introducing the concept of a *current point*, which is selected by the user before she invokes the Draw_line command:

```
Set_attributes   attribute_values   {Only if the current attribute values
                                     are inappropriate}
Select_point     point              {Select start point}
Select_point     point              {Select end point}
Draw_line                           {Draw_line has no parameters—
                                     all have been factored out}
```

Completely parameterless commands are not necessarily desirable. Here, for example, specifying points and then telling the system to connect them by a line eliminates the ability to do rubberband line drawing.

What about applications in which the user tends to perform the same command several times in sequence, each time on different objects? This situation suggests using a *command-mode* syntax in which the command is entered once, followed by an arbitrary number of parameter sets for the command. For instance, the syntax to delete objects could be as follows:

```
Delete_object                       {Establish Delete_object command mode}
object
object
object
.
any command                         {Establish new command mode}
```

Samsung/Google, EX1027, p. 430

Delete_object establishes a deletion mode so that each object selected thereafter is deleted, until any other command is selected. Note that command mode implies a prefix syntax.

If we factor out the object parameter from the command whose syntax is

```
Delete_object        object
```

we introduce the concept of a *currently selected object*, or CSO. We also need a new command, Select_object, to create a mode in which there is a CSO; this CSO can then be operated on by the Delete_object command:

```
Select_object        object        {Use if no object is selected,
                                    or if another CSO is desired}
Delete_object                      {No parameter—the object
                                    has been factored out}
```

The parameter factoring has created a *postfix* syntax: The object is specified first by selection, and then the command is given. This is an especially attractive technique if the user is working with a pointing device, because we have a natural tendency to point at something before saying what to do with it; the converse order is much less natural [SUKA90].

The currently selected object is a useful concept, because the user can perform a series of operations on one object, and then move on to another. Furthermore, the Select_object command usually can be made implicit: The user simply points at the object to be selected and clicks a button. This means that factoring does not need to create extra steps.

Recall that command mode has a prefix syntax. Can its advantages be retained if a postfix syntax is preferred? The answer is yes, *if* a Repeat command, to repeat the last nonselect command, can be made available easily, say by a button-down on a multibutton mouse. If so, the user action sequence to delete several objects could be

```
Select_object        object        {Establish a CSO}
Delete_object                      {Delete the CSO}
Select_object        object        {A new CSO}
Repeat                             {Single button depression to delete}
Select_object        object        {A new CSO}
Repeat                             {Single button depression to delete}
⋮
```

Compare this to the action sequence that would be used with a true command mode:

```
Delete_object
object
object
object
⋮
```

Assuming that Select_object requires just a point and click, the extra steps needed to use Repeat_last_operation with an object mode, as opposed to using a command mode, are a single button-push per deletion.

Samsung/Google, EX1027, p. 431

Another sequencing alternative is the arbitrary *free-form syntax* (*nofix syntax*), which permits intermixing of different syntaxes. Whether the user specifies an object and then an action or an action and then an object, the action is carried out on the object. For example,

```
Set_attributes      attribute values
Select_object       object1             {Attributes applied to object1}
Set_attributes      attribute values
Select_object       object2             {Attributes applied to object2}
Select_object       object3
Set_attributes      attribute values    {Attributes applied to object3}
```

Note that this syntactic structure cannot assume a currently selected object; if it did, then the second Set_attributes command would immediately operate on object1, rather than on object2.

Command and currently selected-object modes can be used with a free-form syntax *if* a Do_it command is added, so the user can tell the system to carry out the current command on the currently selected object. This command does, however, add another user action, as the following sequence illustrates:

```
Select_object       object              {Establish a CSO}
Set_attributes      attribute values    {Establish a current command}
Do_it                                   {CSO receives new attributes}
Copy                                    {Establish a new current command}
Select_object       object              {Establish a new CSO}
Do_it                                   {Copy CSO; copy is now the CSO}
Do_it                                   {CSO copied; new copy is the CSO}
```

An alternative to this free-form syntax is a *mode-sensitive syntax*, which differentiates between the two sequences to make the Set_attributes command mode-sensitive:

```
Set_attributes      attribute values    {No CSO at this point}
Select_object       object
```

and

```
Select_object       object              {Establish a CSO}
Set_attributes      attribute values
```

Mode-sensitivity is a special case of a more general context-sensitivity, by which the effect of a command depends on the current context. In the first of the preceding sequences, where there is no CSO when Set_attributes is used, the attribute values become the global default values that are applied when new objects are created. In the second sequence, where there is a CSO, the attribute values apply to the CSO and do not change the global default values. That is, the existence or nonexistence of a CSO, which is a mode, determines the effect of the command. This technique creates a more powerful set of commands without adding any explicit new commands; however, the user must have mode feedback to know how the command will behave. Also, some users are confused by this approach, because it seems inconsistent until the rules are understood.

Samsung/Google, EX1027, p. 432

The general concept of factoring is important for several reasons. First, new users do not need to be concerned with factored parameters that have default values, which improves learning speed. Values for factored parameters do not need to be specified unless the current values are unacceptable, which improves speed of use. Factoring out the object from the command creates the concept of a CSO, a natural one for interactive graphics with its pointing devices. Finally, factoring reduces or eliminates the short-term modes created by prefix commands with multiple parameters. Factoring has been incorporated into a user-interface design tool so that the designer can request that specific parameters be factored; the necessary auxiliary command (Select_object) is introduced automatically [FOLE89].

There are several variations on the CSO concept. First, when an object is created, it does not need to become the CSO if there is already a CSO. Similarly, when the CSO is deleted, some other object (the most recent CSO or an object close to the CSO) can become the new CSO. In addition, a currently selected set (CSS) made of up several selected objects can be used.

9.5 VISUAL DESIGN

The visual design of a user–computer interface affects both the user's initial impression of the interface and the system's longer-term usefulness. Visual design comprises all the graphic elements of an interface, including overall screen layout, menu and form design, use of color, information codings, and placement of individual units of information with respect to one another. Good visual design strives for clarity, consistency, and attractive appearance.

9.5.1 Visual Clarity

If the meaning of an image is readily apparent to the viewer, we have *visual clarity*. An important way to achieve visual clarity is to use the visual organization of information to reinforce and emphasize the underlying logical organization. There are just a few basic visual-organization rules for accomplishing this end. Their use can have a major influence, as some of the examples will show. These rules, which have been used by graphic designers for centuries [MARC80], were codified by the Gestalt psychologist Wertheimer [WERT39] in the 1930s. They describe how a viewer organizes individual visual stimuli into larger overall forms (hence the term *Gestalt*, literally "shape" or "form," which denotes an emphasis on the whole, rather than on the constituent parts).

The visual-organization rules concern similarity, proximity, closure, and good continuation. The rule of *similarity* states that two visual stimuli that have a common property are seen as belonging together. Likewise, the rule of *proximity* states that two visual stimuli that are close to each other are seen as belonging together. The rule of *closure* says that, if a set of stimuli almost encloses an area or could be interpreted as enclosing an area, the viewer sees the area. The *good-continuation* rule states that, given a juncture of lines, the viewer sees as continuous those lines that are smoothly connected.

Samsung/Google, EX1027, p. 433

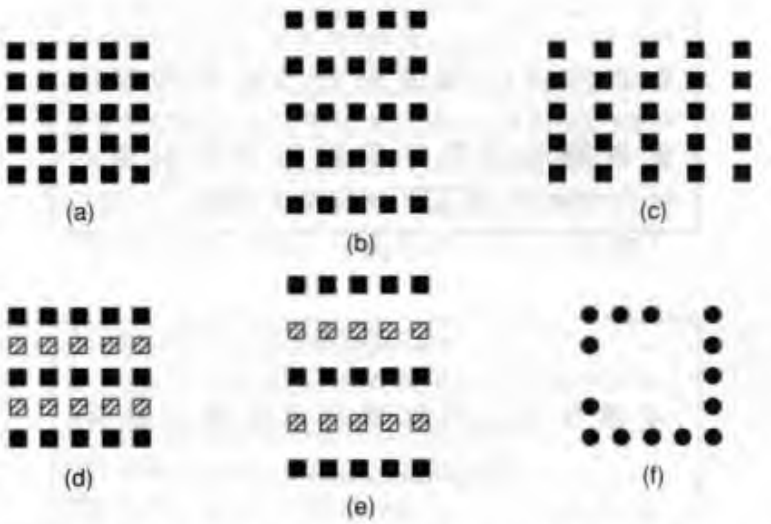


Fig. 9.15 Gestalt rules. In (a), the squares are undifferentiated. In (b), proximity induces a horizontal grouping; in (c), it induces a vertical grouping. In (d), similarity induces a horizontal grouping, which is further reinforced in (e) by a combination of proximity and similarity. In (f), closure induces a square of dots, even though two dots are missing.

Figures 9.15 and 9.16 give examples of these rules and also show how some of them can be combined to reinforce one another. Figure 9.17 shows a form before and after the visual organization rules have been applied. In part (a), everything is near to everything else, so the underlying logical groupings are unclear. Similarity (here, in the sense of being contained in a box) and proximity bind together the patterns and the choice buttons in (b). Closure completes the boxes, which are broken by the label.

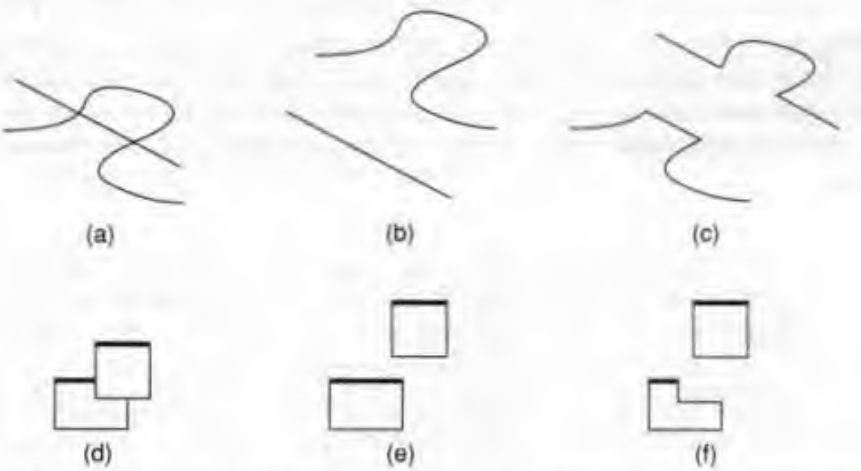
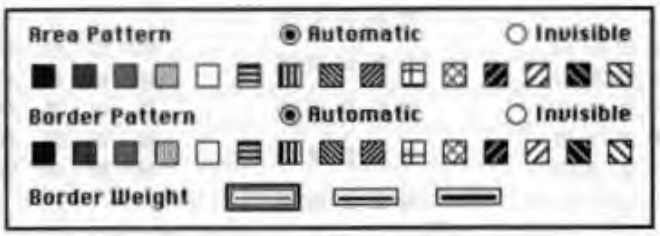
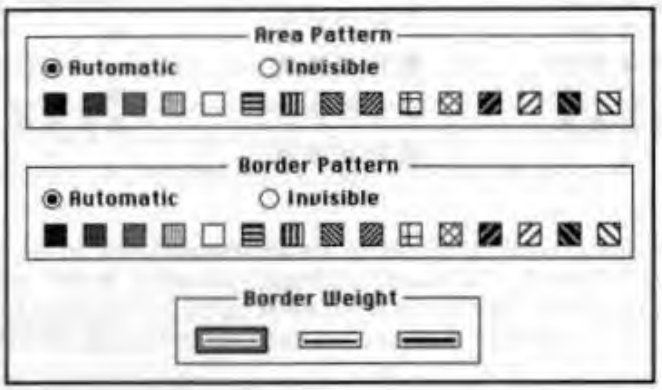


Fig. 9.16 More Gestalt rules. The two intersecting lines in (a) could be interpreted as shown in either (b) or (c). Good continuation favors (b). In a more applied context, the two overlapping windows of (d) could be interpreted as shown in either (e) or (f). Good continuation favors (e).

Samsung/Google, EX1027, p. 434



(a)



(b)

Fig. 9.17 (a) Form laid out with little attention to visual design. (b) Form created using visual grouping and closure to reinforce the logical relationships among the visual elements of the form. (Screen shots © 1983–1989 Microsoft Corporation. Reprinted with permission from Microsoft Corporation.)

The rules are applied to improve menu organization in Fig. 9.18. The leftmost organization is visually unstructured and almost conceals its logical organization. The rightmost menu uses proximity to form groups and similarity of indentation to show the two-level logical structure. In Fig. 9.19, the similarity rule has been used in two different ways (similar typographical style, similar level of indentation) to reinforce logical organization.

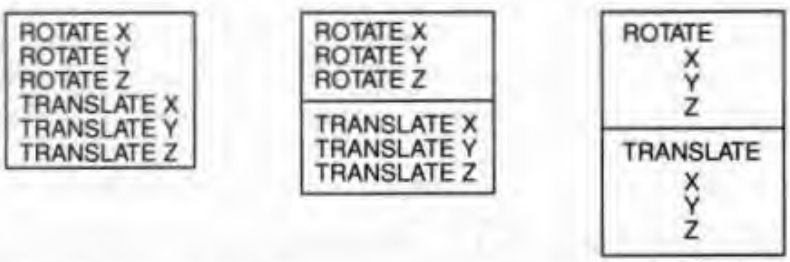


Fig. 9.18 Three designs for the same menu, showing application of visual design techniques.

Samsung/Google, EX1027, p. 435

A MAJOR CATEGORY
A LESS MAJOR CATEGORY
AN EVEN LESS MAJOR CATEGORY
AN EVEN LESS MAJOR CATEGORY
THE LEAST MAJOR CATEGORY
THE LEAST MAJOR CATEGORY
AN EVEN LESS MAJOR CATEGORY

(a)

A MAJOR CATEGORY
A LESS MAJOR CATEGORY
An even less major category
An even less major category
The least major category
The least major category
An even less major category

(b)

A MAJOR CATEGORY
A LESS MAJOR CATEGORY
An even less major category
An even less major category
The least major category
The least major category
An even less major category

(c)

Fig. 9.19 Three designs presenting the same information. (a) The design uses no visual reinforcement. (b) The design uses a hierarchy of typographical styles (all caps boldface, all caps, caps and lowercase, smaller font caps and lowercase) to bond together like elements by similarity. (c) The design adds indentation, another type of similarity, further to bond together like elements.

When ignored or misused, the organization rules can give false visual cues and can make the viewer infer the wrong logical organization. Figure 9.20 gives an example of false visual cues and shows how to correct them with more vertical spacing and less horizontal spacing. Figure 9.21(a) shows a similar situation.

Recall that the objective of using these principles is to achieve visual clarity by reinforcing logical relationships. Other objectives in placing information are *to minimize the eye movements* necessary as the user acquires the various units of information required for a task, and *to minimize the hand movements* required to move a cursor between the parts of the screen that must be accessed for a task. These objectives may be contradictory; the designer's task is to find the best solution.

Samsung/Google, EX1027, p. 436

ATE	BAT	BET
BITE	CAT	CUP
DOG	EAST	EASY
FAR	FAT	FITS
GET	GOT	GUT
HAT	HIGH	HIT

(a)

ATE	BAT	BET
BITE	CAT	CUP
DOG	EAST	EASY
FAR	FAT	FITS
GET	GOT	GUT
HAT	HIGH	HIT

(b)

Fig. 9.20 In (a), the list has a horizontal logical (alphabetical) organization, but a vertical visual organization is induced by the strong proximity relationship. In (b), the alphabetical organization is visually reinforced.

9.5.2 Visual Codings

In interface design, *coding* means creating visual distinctions among several different types of objects. Many different coding techniques are available: color, shape, size or length, typeface, orientation, intensity, texture, line width, and line style are all commonly used in computer graphics. A fundamental issue with any coding technique is to determine how many different categories a particular technique can encode. As more code values are introduced, the possibility of the viewer confusing one value with another increases. The use of a legend, indicating the meaning of each code value, can decrease the error rate.

Many experiments have been conducted to determine how many code values in different coding techniques can be used and still allow almost error-free code recognition (without a legend). For 95-percent error-free performance, 10 colors, 6 area sizes, 6 lengths, 4 intensities, 24 angles, and 15 geometric shapes are the most that can be used [VANC72]. Of course, the code values must be appropriately spaced; see [VANC72, pp. 70–71] for a list of appropriate colors.

If it is important for the viewer to distinguish among different types of information, then it is appropriate to use redundant coding: the use of two different codes to represent the same information. Part (c) of Fig. 9.19 is redundantly coded. Figure 9.22 shows a triply redundant code. Color is normally used redundantly with some other code, to accommodate color-blind users.

Before we can select a code, we must know how many code levels are needed. It is also important to understand whether the information being coded is nominative, ordinal, or ratio. *Nominative* information simply designates, or names, different types of things, such as different types of planes or ships. Nominative information has no notion of greater than or less than. *Ordinal* information is ordered and has a greater than and less than relation. But no metric is defined on ordinal information; there is no notion of varying distances

Samsung/Google, EX1027, p. 437

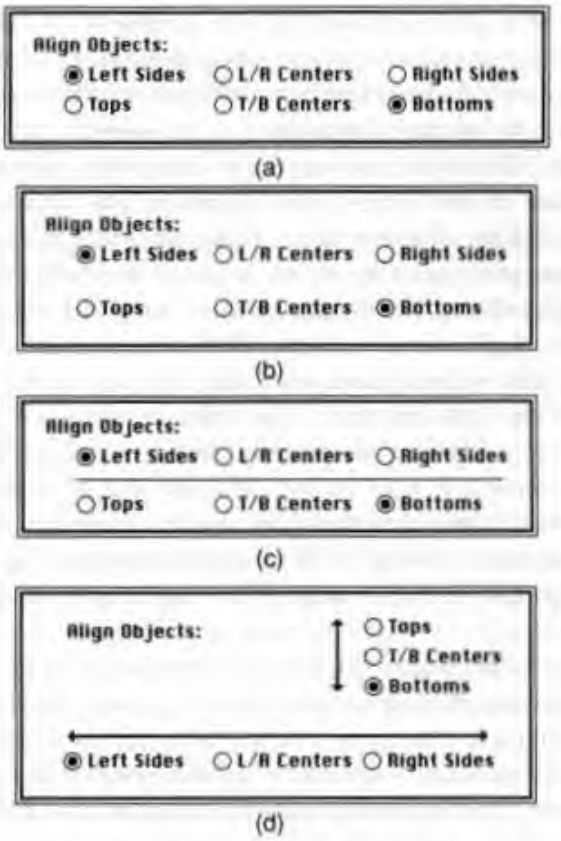


Fig. 9.21 A dialogue box for aligning objects. In (a), the visual cues group the buttons into three groups of two, rather than the proper two groups of three. In (b), the vertical spacing has been increased and the visual cues are correct. In (c), a horizontal rule instead of vertical spacing is used to achieve the same effect in less total space. In (d), the options are rearranged and arrows are used to emphasize the spatial correspondence of each button set to the associated meanings. (Copyright 1988 Claris Corporation. All rights reserved.)

between categories. *Ratio* information has such a metric; examples are temperature, height, weight, and quantity.

For a fixed number of nominative-code values, color is distinguished considerably more accurately than are shape and size, and somewhat more accurately than is intensity [CHRI75]. This suggests that color should be used for coding, but recall that 6 percent to 8



Fig. 9.22 A triply redundant code using line thickness, geometric shape, and interior fill pattern. Any one of the three codes is sufficient to distinguish between the three code values.

Samsung/Google, EX1027, p. 438

percent of males have at least a mild form of color blindness. As discussed in Chapter 13, this need not be a particular problem, especially if redundant coding is used.

Codes used for displaying nominative information should be devoid of any ordering, so that the viewer cannot infer an importance order in the information. Different shapes (such as the squares, circles, diamonds, etc. used for data points when several variables are plotted on the same set of axes), line styles, typefaces, and cross-hatching patterns are appropriate nominative codes. However, using many different typefaces creates a confusing image. A generally accepted guideline is that no more than two or three faces should be used in a single image. Also, differing densities of cross-hatching patterns can create an apparent ordering (Fig. 9.17).

Codes used to display ordinal information can, but need not, vary continuously, but must at least have an obvious ordering. Line styles and area-fill patterns with varying densities can be used, as can text size (many displays provide only a few text sizes, making use of this variable for ratio coding difficult). For both ratio and ordinal information, the apparent visual weight of the codes should increase as the values of the information being coded increase. In Fig. 9.19, ordinal information is being coded, and the typographical hierarchy has a visual weight that increases with the importance of the category.

Ratio information, such as size, length, or orientation, must be presented with a code that can vary continuously. Cleveland and McGill studied the use of several different continuously varying codes to display ratio information, by showing experimental subjects graphs of the same information encoded in different ways. They found the following rankings, where 1 is the most accurately recognized coding [CLEV84; CLEV85]:

1. Position along a common scale
2. Position on identical, nonaligned scales
3. Length
4. Angle between two lines, and line slope
5. Area
6. Volume, density, and color saturation
7. Color hue.

Similarly, Ware and Beatty [WARE88] found that color is effective in grouping objects, but is not effective as a ratio code.

If color were used both to group menu items and to code information in the work area (say, to distinguish layers on a VLSI chip or geographic features on a map), then the user might incorrectly conclude that the red commands in the menu could be applied to only the red elements in the work area. Similarly, a color code might, by using some bright colors and some dark colors, inadvertently imply two logical groupings, one of brighter objects, the other of darker objects. The similarity rule discussed earlier is really at the heart of coding. All like information should be coded with the same code value; all unlike information should have some other code value.

Samsung/Google, EX1027, p. 439

Coding of quantitative data is just a part of the more general field of displaying quantitative data. When data presentations—such as bar, pie, and trend charts—are being designed, many further considerations become important. These are beyond the scope of this text, but are important enough that you should consult key references. The lavishly illustrated books by Bertin [BERT81; BERT83] and Tufte [TUFT83] discuss how to convey quantitative data effectively. Bertin systematically analyzes the visual codes, shows how they can be used effectively, and categorizes different presentation schemes. Tufte argues for minimality in decorative accoutrements to charts and graphs, and for emphasis on the data being conveyed. He also traces the fascinating history of data presentation since 1700. Schmid provides additional guidance [SCHM84].

Mackinlay incorporated some of Bertin's ideas, along with Cleveland and McGill's results into APT, an expert system that automatically creates data presentations [MACK86]. Color Plate I.24 is an example from APT. We expect to see more developments in this promising area.

Closely related to coding are means for calling the viewer's attention to a particular piece of information, such as an error or warning message, the currently selected object, the current command, the failed piece of equipment, or the planes on a collision course. Some attention-getting techniques available are a unique color or shape, a blinking or pulsating or rotating cursor, and reverse video. A unique color was found to be more effective for attracting the viewer's attention than was a unique shape, size, or intensity [CHRI75].

Attention-getting mechanisms can be misused. A pulsating cursor (that is, a cursor whose size continuously varies between large and small) does indeed attract the user's attention. But it also tends to hold attention. When the user is looking at something else on the screen, the pulsating cursor, even though seen only peripherally, is distracting rather than helpful.

Coding of qualitative information is another important research area for user interface design. Work by Feiner and Seligmann [FEIN85; SELI89] explores the automated design of pictures that explain how to perform actions in 3D environments. Based on input about the information the pictures are supposed to communicate and who will be viewing them, a rule-based system determines the objects to include, their properties and rendering style, and the virtual camera parameters that are input to a 3D graphics system that draws the pictures. Color Plate I.25 is an example of a picture generated automatically for a maintenance and repair application.

9.5.3 Visual Consistency

Consistent application of visual-organization rules and codings, and consistent combination of visual elements into higher-level graphic objects and icons, constitute another important element of visual design. Visual consistency is, of course, part of the overall theme of consistency discussed in Section 9.3.1.

Visual elements can be thought of as letters in a graphic alphabet, to be combined into "words" whose meanings should be obvious to the viewer. For instance, dialogue boxes for Macintosh applications are constructed from a small graphic alphabet. Figures 8.16, 8.32,

Samsung/Google, EX1027, p. 440



Fig. 9.23 The graphic alphabet used in many Macintosh applications. The square choice boxes indicate alternatives, of which several may be selected at once. The round choice circles, called "radio buttons," indicate mutually exclusive alternatives; only one may be selected. The rounded-corner rectangles indicate actions that can be selected with the mouse. In addition, the action surrounded by the bold border can be selected with the return key on the keyboard. The rectangles indicate data fields that can be edited. (© Apple Computer, Inc.)

9.14, 9.17, and 9.21 are examples of these dialogue boxes, and Fig. 9.23 shows their graphic alphabet. Similarly, Fig. 9.24 shows the use of a small graphic alphabet to build icons, and Fig. 9.25 shows a single-element graphic alphabet.

Consistency must be maintained among as well as within single images; a consistent set of rules must be applied from one image to another. In coding, for example, it is unacceptable for the meaning of dashed lines to change from one part of an application to another. For placement consistency, keep the same information in the same relative position from one image or screen to the next, so that the user can locate information more quickly.

9.5.4 Layout Principles

Individual elements of a screen not only must be carefully designed, but also, to work together, must all be well placed in an overall context. Three basic layout rules are balance,



Fig. 9.24 (a) A graphics alphabet. (b) Icons formed by combining elements of the alphabet.

Samsung/Google, EX1027, p. 441

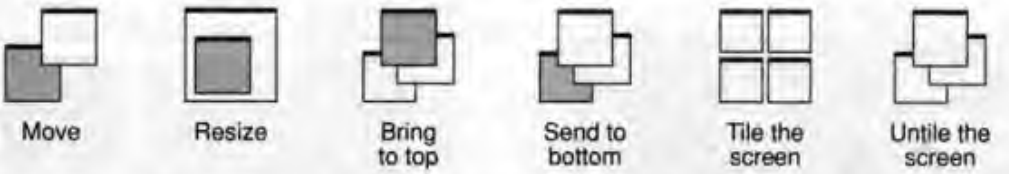


Fig. 9.25 Several different icons, all created from a single shape representing a window.

gridding, and proportion. Figure 9.26 shows two different designs for the same screen. Design (a) is *balanced*, nicely framing the center and drawing the eye to this area. Design (b) is unbalanced, and unnecessarily draws the eye to the right side of the area. Design (b) also has a slight irregularity in the upper right corner: the base lines of the scroll bar arrow and the pointer icon are not quite aligned. The eye is needlessly drawn to such meaningless discontinuities.

Figure 9.27 shows the benefits of using empty space between different areas, and also illustrates the concept of *gridding*; in cases (b) and (c), the sides of the three areas are all aligned on a grid, so there is a neatness, an aesthetic appeal, lacking in (a) and (d). Figure 9.28 further emphasizes the detrimental effects of not using a grid. [FEIN88] discusses an expert system that generates and uses design grids.

Proportion deals with the size of rectangular areas that are laid out on a grid. Certain ratios of the lengths of a rectangle's two sides are more aesthetically pleasing than are others, and have been used since Greco-Roman times. The ratios are those of the square, which is 1:1; of the square root, 1:1.414; of the golden rectangle, 1:1.618; and of the double square, 1:2. The double square is especially useful, because two horizontal double squares can be placed next to a vertical double square to maintain a grid. These and other design rules are discussed in [MARC80; MARC84; PARK88].

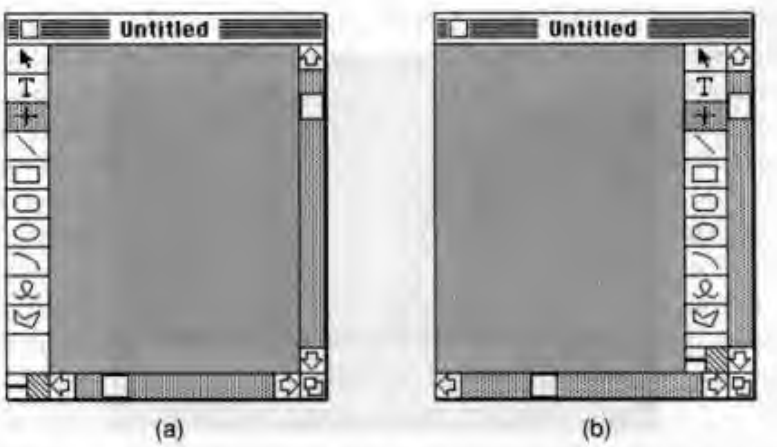


Fig. 9.26 Two alternative screen designs. Design (a) is balanced; design (b) emphasizes the right side. (Copyright 1988 Claris Corporation. All rights reserved.)

Samsung/Google, EX1027, p. 442

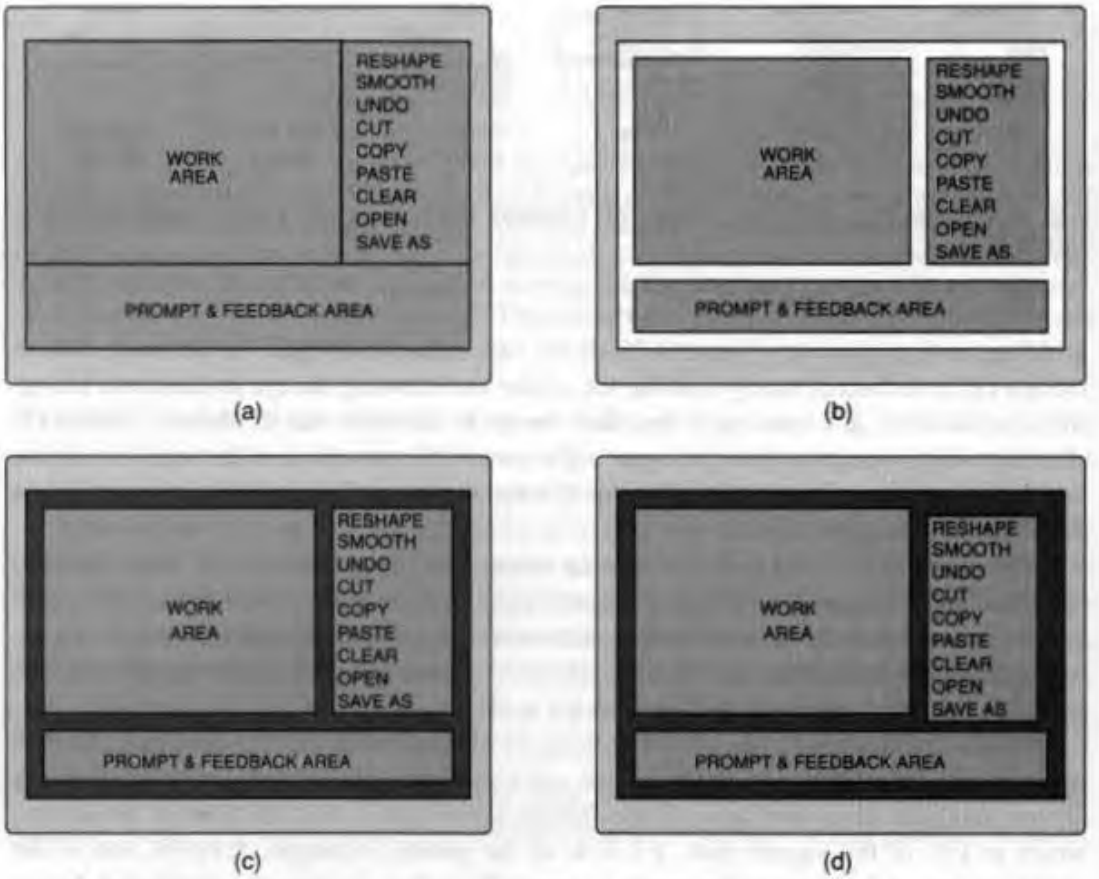


Fig. 9.27 Four screen designs. (a) A typical initial design. (b) Border area has been added. (c) The border has been strengthened to separate the three areas further. (d) The deleterious effect of not aligning elements on a grid is obvious.

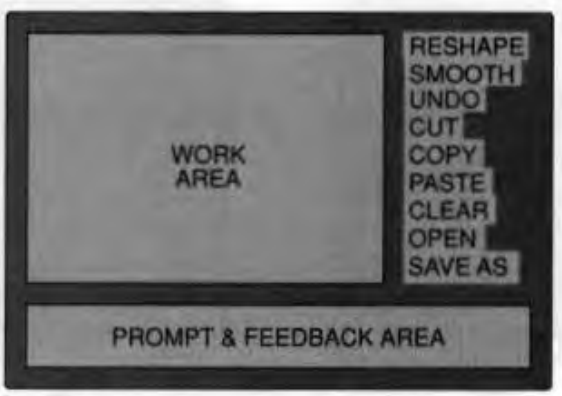


Fig. 9.28 Removing the box around the menu area creates the meaningless but attention-getting ragged-right border.

Samsung/Google, EX1027, p. 443

9.6 THE DESIGN METHODOLOGY

Many ideas have been presented in Chapters 8 and 9. How can a designer integrate them and work with them in a structured way? Although user-interface design is still in part an art rather than a science, we can at least suggest an organized approach to the design process. In this section, we give an overview of the key elements of such a methodology.

The first step in designing an interface is to decide what the interface is meant to accomplish. Although at first this statement may seem trite, poor requirements definitions have doomed numerous user-interface design projects at an early stage. Understanding user requirements can be accomplished in part by studying how the problem under consideration is currently solved. Another successful approach is for the designer to learn how to perform the tasks in question. The objective is to understand what prospective users currently do, and, more important, why they do it.

We do not mean to imply that the interface should exactly mimic current methods. The reason for understanding why prospective users work as they do is often to develop new and better tools. We should recognize, however, that it is sometimes better to mimic old ways to avoid massive retraining or to avoid morale problems with an existing workforce. A typical strategy is first to mimic existing methods, and also to make new methods available; over time, users can be trained in the new or augmented capabilities.

User characteristics must also be identified. What skills and knowledge do the users have? Are the users knowledgable about their work but computer-naive? Are they touch-typists? Will the users typically be eager to learn the system, or will they be reluctant? Will usage be sporadic or regular, full-time or part-time? It is important when assessing the user population to remember that what you, the system designer, would want or like is not necessarily the same as what those for whom the system is being designed might want. Your users are not necessarily created in your own image.

When the requirements have been worked out, a top-down design is next completed by working through the design levels discussed in Section 9.1: conceptual, functional, sequencing, and binding. The rationale for top-down design of user interfaces is that it is best to work out global design issues before dealing with detailed, low-level issues.

The conceptual design is developed first. Ideally, several alternative conceptual designs are developed and evaluated on the basis of how well they will allow users to carry out the tasks identified in the requirements definition. High-frequency tasks should be especially straightforward. Simplicity and generality are other criteria appropriate for the designs.

The functional design focuses on the commands and what they do. Attention must be paid to the information each command requires, to the effects of each command, to the new or modified information presented to the user when the command is invoked, and to possible error conditions. Figure 9.29 shows the focus of functional design. Notice the notations about errors being "engineered out" by subsequent lower-level design decisions. One objective of the functional design is to minimize the number of possible errors, by defining the individual commands appropriately.

The sequencing and binding designs, which together define the form of the interface, are best developed together as a whole, rather than separately. The design involves first selecting an appropriate set of dialogue styles, and then applying these styles to the specific

Samsung/Google, EX1027, p. 444

Function:	Add_symbol_instance
Parameters:	Symbol_identifier Symbol_position
Description:	An instance of the symbol is created and is added to the figure at the desired position. The instance becomes the currently selected object (CSO). The previous CSO, if any, is no longer selected.
Feedback:	The instance is seen on the display and is highlighted because it is selected. (If there was a CSO, it is no longer highlighted.)
Errors:	1. The Symbol_identifier is unknown (engineered out by use of a menu selection to choose symbol). 2. The Symbol_position is outside the viewport (engineered out by constraining the positioning device feedback to be within the viewport).

Fig. 9.29 A typical functional specification for a command. The annotations with the errors are added after interaction techniques are identified as part of designing the form of the interface.

functionality. Sequences of screens, sometimes called *storyboards*, can be used to define the visual and some of the temporal aspects of these designs. State diagrams, as discussed in Section 9.3.1 and in Chapter 8, are also helpful in detailing user-action sequences.

The interface form can be defined by a *style guide*, a written codification of many of the elements of user-interface form. The most common motivation for developing a style guide is to ensure a "look and feel" consistency within and among applications. Some elements of the style guide can be implemented in libraries of interaction techniques (Chapter 10); other elements must be accommodated by the designers and programmers. Many style guides exist, among them guides for the Macintosh [APPL87], Open Software Foundation's OSF/MOTIF [OPEN89], NASA's Transportable Application Executive [BLES88b], and DEC's XUI [DIGI89].

The whole design process is greatly aided by interleaving design with user-interface prototyping. A user-interface prototype is a quickly created version of some or all of the final interface, often with very limited functionality. The emphasis is on speedy implementation and speedy modifiability. At the start, some design questions will seem to be unanswerable; once a prototype is available, however, the answers may become apparent. Prototyping is often superior to using a design document, since it gives users a more specific frame of reference, within which they can talk about their needs, likes, and dislikes. HyperCard and Smalltalk are used extensively for rapid prototyping, as are some of the software tools discussed in Section 10.6.

Prototyping can begin as soon as a conceptual design is worked out, and the elements of the functional design and dialogue style can be developed concurrently. It is important to

Samsung/Google, EX1027, p. 445

follow the Bauhaus dictum, *form follows function*, lest the user-interface style dictate the capabilities of the overall system. As soon as even some modest elements of the interface are developed, potential users should be exposed to them, to elicit suggestions for improvements to the interface. As modifications are made and as the prototype becomes more comprehensive, users should again work with the system. This iterative cycle has come to be viewed as essential to the development of high-quality user-interface software. Further discussion of prototyping and iterative development can be found in [ALAV84; HART89].

EXERCISES

9.1 Determine how several commands in an interactive graphics application program with which you are familiar could be made into direct-manipulation operations.

9.2 Examine several interactive graphics application programs and characterize their dialogue style. List the ways in which the interfaces do and do not follow the design guidelines discussed in this chapter. Identify the design level for each point you list; for instance, consistent use of color is at the hardware-binding level.

9.3 The conceptual model for many word processors is partially based on an analogy with typewriters. List ways in which this analogy might create difficulties for a user who attempts to carry it further than is realistic.

9.4 Analyze a user interface to determine what methods, if any, are provided for error correction. Categorize the methods according to the four types discussed in Section 9.3.4.

9.5 What is the form of the state diagram representing a completely modeless interface?

9.6 Design and implement a simple graphics editor with the following functionality: create, delete, and move lines; move endpoints of lines; change the line style (dash, dotted, solid) of existing lines; set the line-style mode for lines that have not yet been created. Design this system to support two or more of the following five syntaxes, and include a command to switch from one syntax to another: object mode, command mode, object mode with Repeat_last_operation operation, free-form syntax, and free-form syntax with modes and a Do_it command.

9.7 Conduct either informal or formal controlled experiments with each of the syntaxes implemented in Exercise 9.6. Test ease of learning and speed of use. Give your users five predefined tasks to perform. Identify tasks that will be faster to perform with one syntax than with the others.

9.8 Study three different interactive graphics application programs.

a. Identify the classes of modes and syntaxes used in each, using the definitions in Section 9.4. A single application may have several types of modes or syntaxes. If so, are there clear distinctions showing which to use when?

b. Identify the factoring, if any, that has occurred. Are there additional possibilities for factoring? Do you think the user interface would be improved by application of the factoring?

9.9 Consider the three window organizations shown in Fig. 9.30. Count the number of mouse movements and mouse-button depressions needed to move the lower-right corner. In each case, assume that the cursor starts in the center of the window and must be returned to the center of the window. Counting mouse movements as 1.1 seconds and button depressions as 0.2 seconds, how long does each window organization take? Does this result mean that one organization is better than the others?

Samsung/Google, EX1027, p. 446

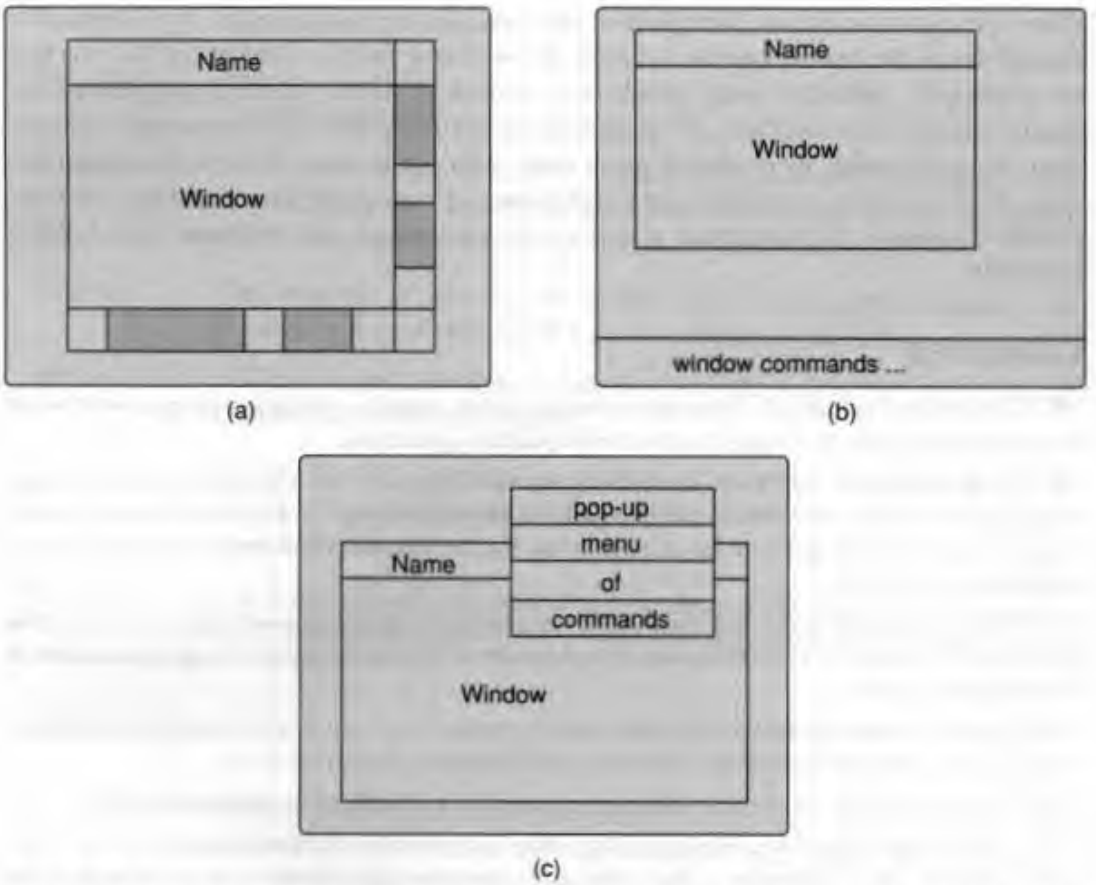


Fig. 9.30 Three means to invoke window commands. In (a) (Macintosh style), regions of the window dressing are used. To resize, the user selects and drags the lower-right corner. In (b), the commands are in the command area at the bottom of the screen. To resize, the user selects a command, and then drags the lower-right corner of the window. In (c), a pop-up menu appears at the cursor when a mouse button is pressed. To resize, the user selects the command from the pop-up menu, and drags the lower-right corner of the window.

9.10 Implement a single-level Undo command for an interactive graphics application program you have written. Decide which of several implementation strategies you will use, and justify your choice. The strategies include (1) after each command, make a complete copy of the application data structure, state variables, and so on; (2) save a record of what was changed in the application data structure; (3) save all the commands since logon, replay them to effect undo; (4) save the application data structure every 10 minutes, plus save all commands since the last such save operation; and (5) save, for each user command, the one or several commands needed to undo the user command. Would your choice differ for a multi-level Undo command?

9.11 Examine an interactive graphics application. How many commands does it have? List the "starter kit" of commands with which a new user can do a simple piece of work. How big is the starter kit with respect to the overall command set? List the ways in which defaults or other methods have been used to minimize the complexity of the starter kit. In your opinion, does the system have a good starter kit?

Samsung/Google, EX1027, p. 447

9.12 Explore other ways of redesigning Fig. 9.17(a) to reinforce the logical relationships of the visual elements.

9.13 Study the dialogue-box design of an application. Which methods are used to reinforce logical structure with visual structure? Can you further improve the design?

9.14 Figure 9.6 shows 12 icons that represent Macintosh programs: (a) disk copy utility, (b) resource mover, (c) icon editor, (d) menu editor, (e) alert/dialog editor, (f) edit program, (g) boot configure, (h) switcher, (i) examine file, (j) MacWrite, (k) MacDraw, and (l) MacPaint. Some of the icons indicate the associated program better than others do.

a. Design an alternative set of icons to represent the 12 programs.
b. Show the set of icons to 10 programmers who are not familiar with the Macintosh, and ask them to guess what each icon means. Tabulate the results.
c. Tell 10 programmers who are not familiar with the Macintosh what each icon means. Give them 2 minutes to study the icons. Ten minutes later, show them the icons again, and ask them what the icons mean. Tabulate the results.
d. Repeat parts (b) and (c) for the icons in the figure. What conclusions can you draw from your data?

9.15 Analyze the visual design of a graphics application. What is the visual alphabet? What visual codings are used? What visual hierarchies are established? Redesign some of the visual cues to emphasize further the underlying logical relationships.

9.16 List 10 specific examples of codings in computer graphics applications you have seen. Is the coded information nominative, ordinal, or ratio? Are the code values appropriate to the information? Are there any false codings?

9.17 Examine three different window managers. What visual code indicates which window is the "listener"—that is, the window to which keyboard input is directed? What visual code indicates which processes are active, as opposed to blocked? What other visual codings are used?

Samsung/Google, EX1027, p. 448

10 User Interface Software

The first two chapters on user interfaces, Chapters 8 and 9, concentrated on the external characteristics of user–computer interfaces. Here, we examine the software components, beyond the basic graphics packages already discussed, that are used in implementing interfaces. Figure 10.1 shows the various levels of user-interface software, and suggests the roles for each. The figure shows that the application program has access to all software levels; programmers can exploit the services provided by each level, albeit with care, because calls made to one level may affect the behavior of another level. The operating-system level is not discussed in this text, and the basics of graphics subroutine packages have already been described. Some input features of device-independent graphics subroutine packages are compared and evaluated in Section 10.1. Window-management systems, discussed in Sections 10.2 to 10.4, manage the resources of screen space and interaction devices so that several applications or multiple views of the same application can share the display. Some window-management systems have an integral graphics subroutine package that provides device-independent abstractions, whereas others simply pass graphics calls through to the underlying graphics hardware or software.

The interaction techniques discussed in Chapter 8 are useful in many applications, but require careful development to provide a pleasing look and feel. Interaction-technique toolkits, treated in Section 10.5, are built on window-management systems to give the application developer a common set of techniques. The final layer, the user-interface management system (UIMS), discussed in Section 10.6, provides additional generic user-interface support at the sequencing level of design (Section 9.1). UIMSs speed up implementation of a user interface, and facilitate making rapid changes during the interface-debugging process discussed in Section 9.6.

Samsung/Google, EX1027, p. 449

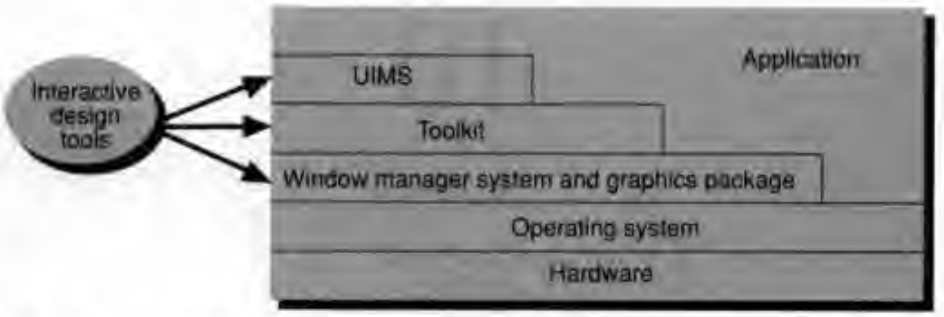


Fig. 10.1 Levels of user-interface software. The application program has access to the operating system, window-manager system and graphics package, toolkit, and user-interface management system (UIMS). The interactive design tools allow nonprogrammers to design windows, menus, dialogue boxes, and dialogue sequences.

10.1 BASIC INTERACTION-HANDLING MODELS

In this section, we elaborate on the interaction-handling capabilities of contemporary device-independent graphics subroutine packages, as introduced in Chapters 2 (SRGP) and 7 (SPHIGS). The sampling and event-driven processing in these two packages is derived from GKS [ANSI85b; ENDE86] and PHIGS [ANSI88], which share a common interaction model. Window-management systems use an event mechanism similar to, but more powerful than, the GKS/PHIGS model discussed in this section.

GKS and PHIGS have six classes of logical input devices, and there may be more than one device of each class associated with a workstation (a display and associated interaction devices). Each of the logical input devices can operate in one of three modes: sample, request, and event. Each device has an associated *measure*, which is the type of information returned by the device. The devices and their measures are as follows:

Device	Measure
locator	position in world coordinates
pick	pick path for SPHIGS, segment identification for GKS
choice	integer indicating the choice
valuator	real number
string	character string (called keyboard device in SRGP and SPHIGS)
stroke	sequence of positions in world coordinates

SRGP and SPHIGS use measures that are slightly different from these.

In *request mode*, the application program requests input from a device, and the graphics package returns control and the measure of the device only after the user has performed an action with the device. The action is called the *trigger*. The specific trigger action for each logical device class is implementation-dependent, but is typically a button-push. For instance, a mouse button triggers locator or pick devices, and the return key triggers the string device.

Request mode can be used with only one device at a time, and is intended to support the limited functionality of older graphics terminals, which are typically connected to computers via RS-232 interfaces. Interaction techniques such as keyboard accelerators

Samsung/Google, EX1027, p. 450

cannot be used, because the application program must know in advance from which device to request input. In addition, as the measure is modified (by moving the mouse to change the locator's measure, say), the application program generally cannot provide dynamic feedback, because the application program does not regain control until the trigger action occurs. This difficulty can be eliminated by defining the trigger action as a small change in the measure.

In *sample mode*, a single device is sampled, and the measure of the device is immediately returned. We can permit the user to select one of several devices to use, by polling all eligible devices, as follows:

```
terminate = FALSE;
while (!terminate) {
    SamplePick (&status, &segmentName);
    /* status = OK means had successful pick; segmentName is identification */
    /* of picked item */
    Process pick input
    SampleString (string);
    Process string input
    /* terminate set to TRUE as part of processing string or pick */
}
```

Sampling in this way is dangerous, however. If the user makes several more inputs while the first is being processed, they will never be seen by the application program, since it stops sampling while processing the first input. Also, the sequence of user events, which is often essential to maintain, might be lost in sampling. Unlike request mode, however, sample mode is well suited for dynamic feedback from the application program, because no trigger action is required to return the device measure to the application program.

Event mode avoids the problems of sample and request modes, by allowing input to be accepted asynchronously from several different devices at once. As discussed in Section 2.3.6 of Chapter 2, the application program first enables all devices whose use is to be permitted. Once the devices are enabled, a trigger action for any of them places an event report on an input queue, in order of occurrence. As seen in Fig. 10.2, the application

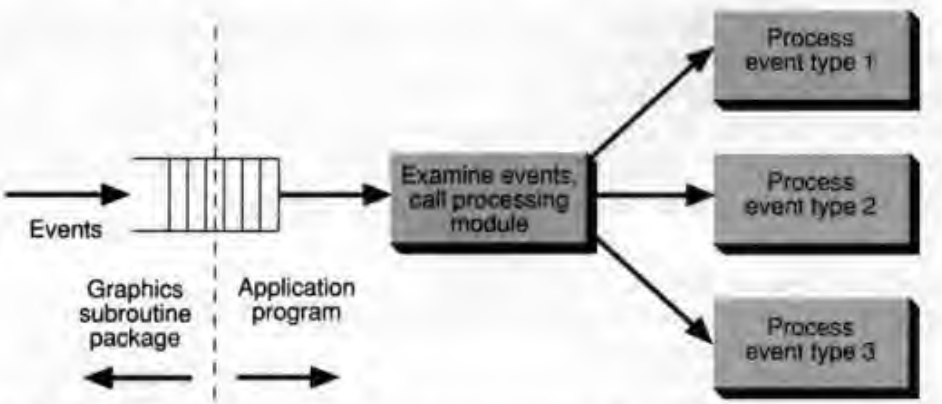


Fig. 10.2 The application program removes events from the queue and dispatches control to the appropriate procedure, which processes them.

Samsung/Google, EX1027, p. 451

program checks the queue to see what user actions have actually occurred, and processes the events as they are removed from the queue.

The following code fragment reimplements the previous polling example in event mode:

```
terminate = FALSE;
while (!terminate) {
    WaitEvent (timeout, &deviceClass, &deviceId);          /* Wait for user action */
    switch (deviceClass) {
        case pick: Process pick;
            break;
        case string: Process string;
            break;
    } /* terminate set to TRUE in processing of pick or string */
}
```

Unlike request-mode input, event-mode input is asynchronous: Once a device is enabled, the application program can be executing while the user is concurrently inputting information with the enabled input devices. This is sometimes called *typeahead*, or, when done with a mouse, *mouseahead*.

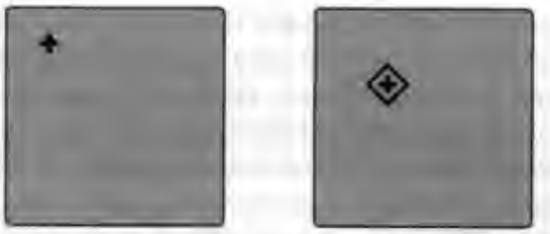
The typeahead capability of the event-queue mechanism provides an opportunity to speed up interactions with the computer. Suppose a button-press (choice logical device) is used to scroll through a drawing. Each button-press scrolls, say, x inches. If the user presses the button more rapidly than the scroll can occur, events build up in the queue. The application program can look for multiple successive button events on the queue; if there are n such events, then a single scroll of nx inches can be performed, and will be much faster than n scrolls of x inches each.

Care must be taken to manage the event queue properly. If the first of two events on the queue causes the program to enable a different set of logical input devices and then to call WaitEvent, the program now may not be expecting the second event, leading to unexpected results. The call FlushDeviceEvents is provided to alleviate this problem; the application program can empty the event queue to ensure that the queue contains nothing unexpected. However, flushing the queue may leave the user wondering why the second event was never processed.

Another concern with event queues is a possible time delay between when an event occurs and when other information needed in connection with that event is obtained. Suppose we want a mouse button-down to display a diamond at the mouse position. If the buttons on the mouse can be assigned to a logical choice device and the (x, y) coordinates of the mouse to a logical locator (this contrasts with the SRGP locator, whose measure includes button status), then we can use the following approach:

```
WaitEvent (timeout, &deviceClass, &deviceId);
if (deviceClass == CHOICE && deviceId == 1) {
    SampleLocator (MOUSE, x, y);                           /* Get position of diamond */
    DrawDiamond (x, y);                                    /* Draw diamond at (x, y) */
}
```

Samsung/Google, EX1027, p. 452



At time t_1, the user makes a choice, with a choice device, to draw a diamond. The user then moves cursor to go on to another task.

At time t_2, locator is sampled and diamond is drawn. Locator has moved between t_1 and t_2.

Fig. 10.3 The effect of a time delay between an event and sampling of the cursor position associated with that event.

The problem is that, between time t_1 (when the WaitEvent procedure returns) and time t_2 (when the SampleLocator procedure obtains the (x, y) coordinates), seconds may have elapsed. In this time, the user may have moved the locator some distance, causing the unexpected result shown in Fig. 10.3. Substantial delays can easily occur on a time-shared computer if another program takes control of the processor, and on any computer that supports virtual memory when page-fault interrupts occur. Thus, although we would like the application program to be uninterruptible during the interval from t_1 to t_2, we cannot guarantee that it will be so.

In GKS and PHIGS, the risk of this time delay can be reduced and even eliminated by activating several logical devices with the same trigger. If we define the trigger to be a button-click on any of the three buttons, and associate this trigger with both the three-button choice device and the locator device, then both events will be placed on the queue (in unspecified order) at the same time. The device-dependent driver under the graphics package can do this faster than the application program can execute the preceding code segment, so the likelihood of being interrupted is less. If the operating system grants the device driver the privilege of disabling interrupts, then there will never be a time delay.

Unfortunately, some user-interface constructs are difficult to provide by means of these logical-device concepts. For instance, it is convenient to use time intervals to distinguish between two commands. Thus, we can select an icon with a single mouse button-click while the cursor is on the icon, and then open the icon with a second click within some small Δt of the first click. Making these distinctions requires that the event reports have a *timestamp* giving the time at which the event occurred. This concept is not found in GKS and PHIGS, although a timestamp could be provided readily.

10.2 WINDOW-MANAGEMENT SYSTEMS

A *window-management system* provides many of the important features of modern user-computer interfaces: applications that show results in different areas of the display, the ability to resize the screen areas in which those applications are executing, pop-up and pull-down menus, and dialogue boxes.

Samsung/Google, EX1027, p. 453