Computer Graphics

PRINCIPLES AND PRACTICE

Foley • van Dam • Feiner • Hughes

SECOND EDITION in C

THE SYSTEMS PROGRAMMING SERIES

TEXAS INSTRUMENTS EX. 1009 - 1/1253

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`

TEXAS INSTRUMENTS EX. 1009 - 2/1253

THE
SYSTEMS
PROGRAMMING
SERIES

Computer Graphics: Principles and Practice

SECOND EDITION in C

Foley • van Dam • Feiner • Hughes

TEXAS INSTRUMENTS EX. 1009 - 3/1253

Plate A "Lilac the Bear," by J. Kajiya, T. Kay, and J. Snyder. (Produced at Caltech and IBM Corporation. Copyright © 1989 Caltech.)

Plate C "Voxel Garden." (Courtesy of Ned Greene, NYIT Computer Graphics Lab.)

Plate B "Blessed State," by F. K. Musgrave. (Copyright © 1989 K. Musgrave and B. Mandelbrot.)

Plate D "Luxo Jr.," by J. Lasseter, W. Reeves, E. Ostby, and S. Leffler. (Copyright © 1986 Pixar.)

SECOND EDITION IN C

Computer Graphics

PRINCIPLES AND PRACTICE

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

TEXAS INSTRUMENTS EX. 1009 - 7/1253

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

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

Library of Congress Cataloging-in-Publication Data

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

Reprinted with corrections, July 1997.

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

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

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

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

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

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

6 7 8 9 10-DOC-99 98 97

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.

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

TEXAS INSTRUMENTS EX. 1009 - 10/1253

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

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 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

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 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 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 &

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.

Contents

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

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

1.1 IMAGE PROCESSING AS PICTURE ANALYSIS

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

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

1.2 THE ADVANTAGES OF INTERACTIVE GRAPHICS

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

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

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

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

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

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

1.3 REPRESENTATIVE USES OF COMPUTER GRAPHICS

Computer graphics is used today in many different areas of industry, business, government, education, entertainment, and, most recently, the home. The list of applications is enormous and is growing rapidly as computers with graphics capabilities become commodity products. Let's look at a representative sample of these areas.

- *User interfaces*. As we mentioned, most applications that run on personal computers and workstations, and even those that run on terminals attached to time-shared computers and network compute servers, have user interfaces that rely on desktop window systems to manage multiple simultaneous activities, and on point-and-click facilities to allow users to select menu items, icons, and objects on the screen; typing is necessary only to input text to be stored and manipulated. Word-processing, spreadsheet, and desktop-publishing programs are typical applications that take advantage of such user-interface techniques. The authors of this book used such programs to create both the text and the figures; then, the publisher and their contractors produced the book using similar typesetting and drawing software.
- *(Interactive) plotting in business, science, and technology*. The next most common use of graphics today is probably to create 2D and 3D graphs of mathematical, physical, and economic functions; histograms, bar and pie charts; task-scheduling charts; inventory and production charts; and the like. All these are used to present meaningfully and concisely the trends and patterns gleaned from data, so as to clarify complex phenomena and to facilitate informed decision making.
- *Office automation and electronic publishing*. The use of graphics for the creation and dissemination of information has increased enormously since the advent of desktop publishing on personal computers. Many organizations whose publications used to be printed by outside specialists can now produce printed materials inhouse. Office automation and electronic publishing can produce both traditional printed (hardcopy) documents and electronic (softcopy) documents that contain text, tables, graphs, and other forms of drawn or scanned-in graphics. Hypermedia systems that allow browsing of networks of interlinked multimedia documents are proliferating (see Color Plate I.2).
- *Computer-aided drafting and design*. In computer-aided design (CAD), interactive graphics is used to design components and systems of mechanical, electrical, electromechanical, and electronic devices, including structures such as buildings, automobile bodies, airplane and ship hulls, very large-scale-integrated (VLSI) chips, optical systems, and telephone and computer networks. Sometimes, the user merely wants to produce the precise drawings of components and assemblies, as for online drafting or architectural blueprints. Color Plate I.8 shows an example of such a 3D design program, intended for nonprofessionals: a "customize your own patio deck" program used in lumber yards. More frequently, however, the emphasis is on interacting with a computer-based model of the component or system being designed in order to test, for example, its structural, electrical, or thermal properties. Often, the model is interpreted by a simulator that feeds back the behavior of the system to the user for further interactive design and test cycles. After objects have been designed, utility programs can *postprocess* the design database to make parts lists, to process "bills of materials," to define numerical control tapes for cutting or drilling parts, and so on.
- *Simulation and animation for scientific visualization and entertainment*. Computer-produced animated movies and displays of the time-varying behavior of real and simulated

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

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

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

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

1.4 CLASSIFICATION OF APPLICATIONS

The diverse uses of computer graphics listed in the previous section differ in a variety of ways, and a number of classifications may be used to categorize them. The first classification is by *type (dimensionality) of the object* to be represented and the *kind of picture* to be produced. The range of possible combinations is indicated in Table 1.1.

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

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

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

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

TABLE 1.1 CLASSIFICATION OF COMPUTER GRAPHICS BY OBJECT AND PICTURE

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

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

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

1.5 DEVELOPMENT OF HARDWARE AND SOFTWARE FOR COMPUTER GRAPHICS

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

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

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

Since at that time computer input/output (I/O) was done primarily in batch mode using punched cards, hopes were high for a breakthrough in interactive user–computer communication. Interactive graphics, as "the window on the computer," was to be an integral part of vastly accelerated interactive design cycles. The results were not nearly so dramatic, however, since interactive graphics remained beyond the resources of all but the most technology-intensive organizations. Among the reasons for this were these:

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

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

1.5.1 Output Technology

The display devices developed in the mid-sixties and in common use until the mid-eighties are called *vector*, *stroke*, *line drawing*, or *calligraphic displays*. The term *vector* is used as a synonym for *line* here; a *stroke* is a short line, and *characters* are made of sequences of such strokes. We shall look briefly at vector-system architecture, because many modern raster graphics systems use similar techniques. A typical vector system consists of a display processor connected as an I/O peripheral to the central processing unit (CPU), a display buffer memory, and a CRT. The buffer stores the computer-produced *display list* or *display program*; it contains point- and line-plotting commands with (x, y) or (x, y, z) endpoint coordinates, as well as character-plotting commands. Figure 1.1 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

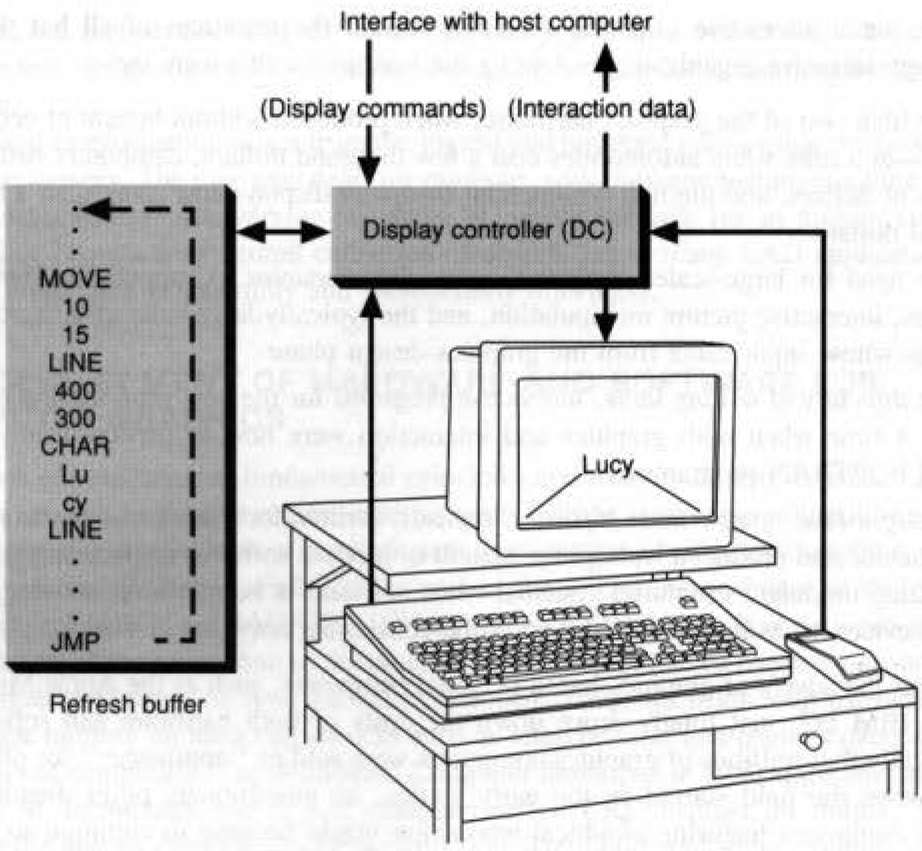


Fig. 1.1 Architecture of a vector display.

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

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

Another major hardware advance of the late sixties was attaching the display to a minicomputer; with this configuration, the central time-sharing computer was relieved of the heavy demands of refreshed display devices, especially user-interaction handling, and updating the image on the screen. The minicomputer typically ran application programs as well, and could in turn be connected to the larger central mainframe to run large analysis programs. Both minicomputer and DVST configurations led to installations of thousands of graphics systems. Also at this time, the hardware of the display processor itself was becoming more sophisticated, taking over many routine but time-consuming jobs from the graphics software. Foremost among such devices was the invention in 1968 of refresh display hardware for geometric transformations that could scale, rotate, and translate points and lines on the screen in real time, could perform 2D and 3D clipping, and could produce parallel and perspective projections (see Chapter 6).

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

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

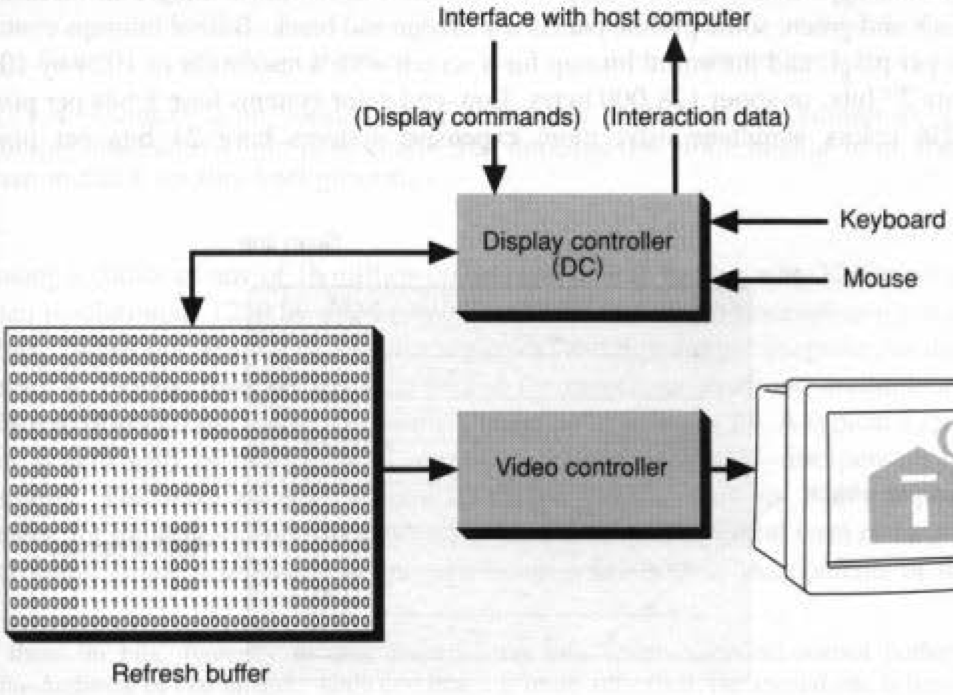


Fig. 1.2 Architecture of a raster display.

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

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

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

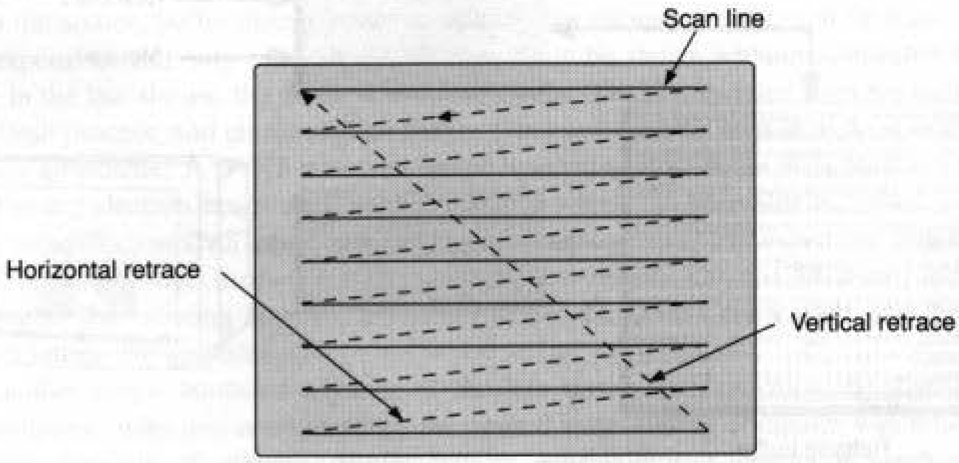


Fig. 1.3 Raster scan.

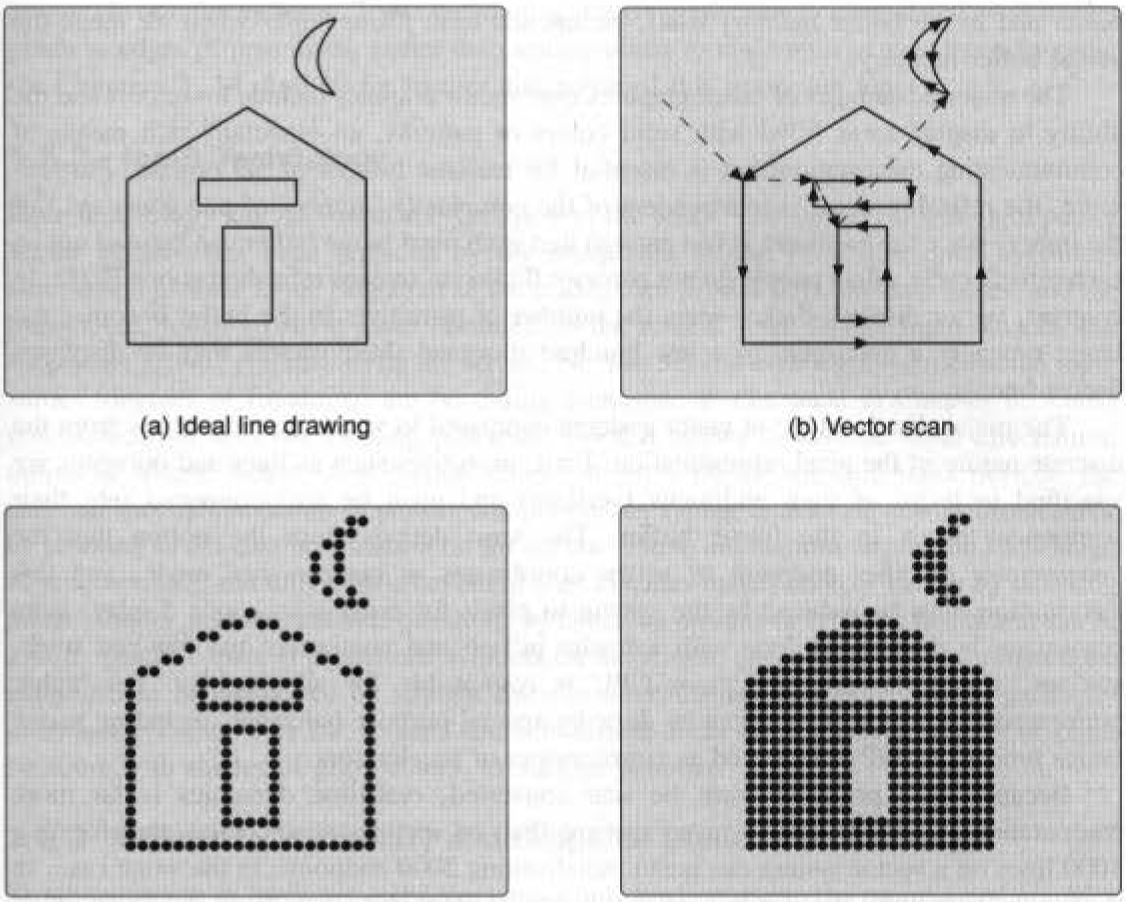

(a) Ideal line drawing
(b) Vector scan
(c) Raster scan with outline primitives
(d) Raster scan with filled primitives

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

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

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

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

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

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

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

The second drawback of raster systems arises from the nature of the raster itself. Whereas a vector system can draw a continuous, smooth line (and even some smooth curves) from essentially any point on the CRT face to any other, the raster system can display mathematically smooth lines, polygons, and boundaries of curved primitives such as circles and ellipses only by approximating them with pixels on the raster grid. This can cause the familiar problem of "jaggies" or "staircasing," as shown in Fig. 1.4 (c) and (d). This visual artifact is a manifestation of a sampling error called *aliasing* in signal-processing theory; such artifacts occur when a function of a continuous variable that contains sharp changes in intensity is approximated with discrete samples. Both theory and practice in modern computer graphics are concerned with techniques for *antialiasing* on gray-scale or color systems. These techniques specify gradations in intensity of neighboring pixels at edges of primitives, rather than setting pixels to maximum or zero intensity only; see Chapters 3, 14, and 19 for further discussion of this important topic.

1.5.2 Input Technology

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

1.5.3 Software Portability and Graphics Standards

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

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

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

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

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

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

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

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

1.6 CONCEPTUAL FRAMEWORK FOR INTERACTIVE GRAPHICS

1.6.1 Overview

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

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

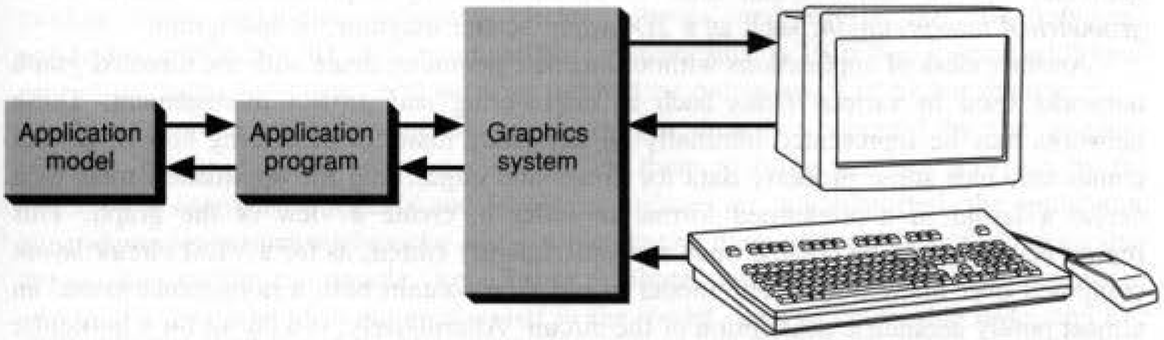


Fig. 1.5 Conceptual framework for interactive graphics.

1.6.2 Application Modeling

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

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

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

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

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

1.6.3 Describing to the Graphics System What Is to Be Viewed

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

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

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

If the model stores geometric primitives not directly supported by the graphics package, the application program must reduce them to those that are accepted by the system. For example, if spheres and free-form surfaces are not supported, the application must approximate such surfaces by *tiling* or *tessellating* them with meshes of polygons that the graphics system can handle. Appearance attributes supported by the graphics package also tend to correspond to the ones stored in the model, such as color, line style, and line width. In addition to primitives and attributes, advanced graphics packages such as PHIGS support facilities for specifying geometric transformations for scaling, rotating, and positioning components, for specifying how components are to be viewed in 3D, and for grouping logically related primitives, attributes, and transformations so that they can be invoked by a single reference to a named structure.

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

1.6.4 Interaction Handling

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

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

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

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

1.7 SUMMARY

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

Until the late eighties, the bulk of computer-graphics applications dealt with 2D objects; 3D applications were relatively rare, both because 3D software is intrinsically far more complex than is 2D software and because a great deal of computing power is required to render pseudorealistic images. Therefore, until recently, real-time user interaction with 3D models and pseudorealistic images was feasible on only very expensive high-performance workstations with dedicated, special-purpose graphics hardware. The spectacular progress of VLSI semiconductor technology that was responsible for the advent of inexpensive microprocessors and memory led in the early 1980s to the creation of 2D, bitmap-graphics–based personal computers. That same technology has made it possible, less than a decade later, to create subsystems of only a few chips that do real-time 3D animation with color-shaded images of complex objects, typically described by thousands of polygons. These subsystems can be added as 3D accelerators to workstations or even to personal computers using commodity microprocessors. It is clear that an explosive growth of 3D applications will parallel the current growth in 2D applications. Furthermore, topics such as photorealistic rendering that were considered exotic in the 1982 edition of this book are now part of the state of the art and are available routinely in graphics software and increasingly in graphics hardware.

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

EXERCISES

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

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

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

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

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

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

1.7 Consider an image containing a set of 10,000 1-inch unconnected vectors. Contrast the storage required for a vector display list with that for a 1-bit raster image for a 1024-by-1024 bilevel display to store this image. Assume that it takes a 8-bit "op-code" to specify "vector-draw," and four 10-bit coordinates (i.e., 6 bytes) to store a vector in the vector display list. How do these numbers vary as a function of the number and size of the vectors, the number of bits per pixel, and the resolution of the raster display? What conclusion can you draw about the relative sizes of refresh memory required?

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

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

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 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.

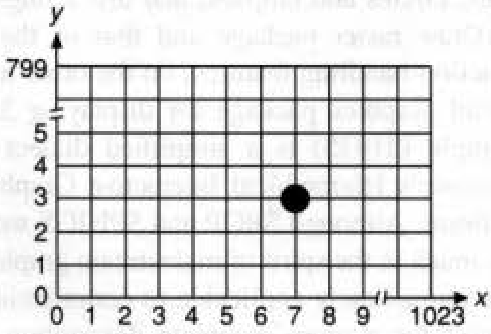


Fig. 2.1 Cartesian coordinate system of a screen 1024 pixels wide by 800 pixels high. Pixel (7, 3) is shown.

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.

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

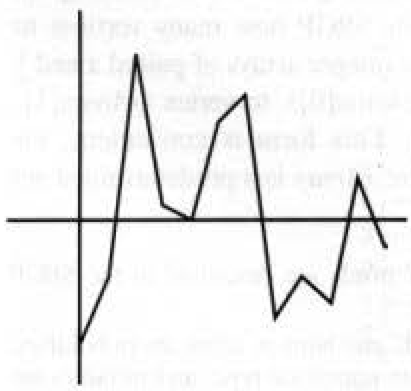
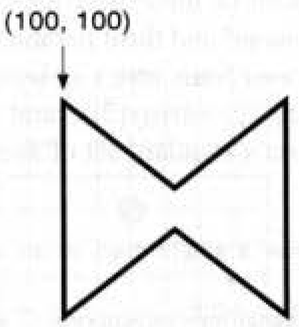


Fig. 2.2 Graphing a data array.



	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.

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;
```

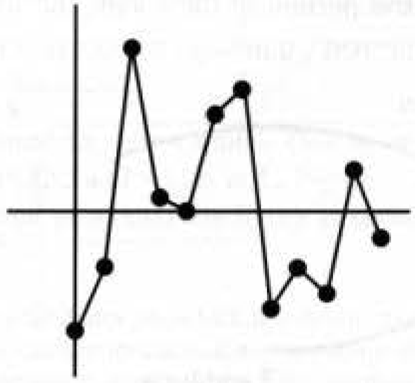


Fig. 2.4 Graphing the data array using markers.

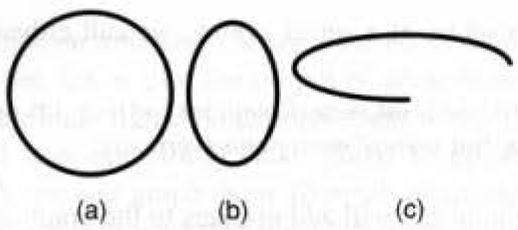


Fig. 2.5 Ellipse arcs.

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

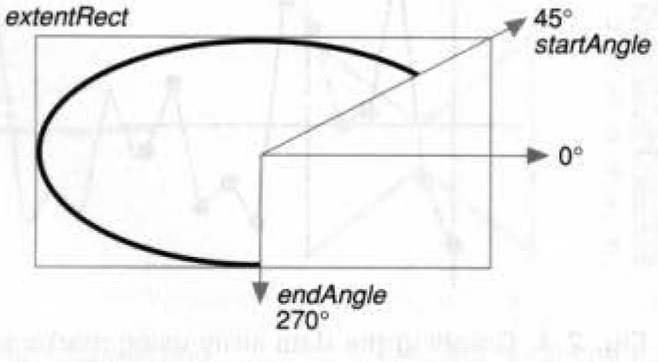


Fig. 2.6 Specifying ellipse arcs.

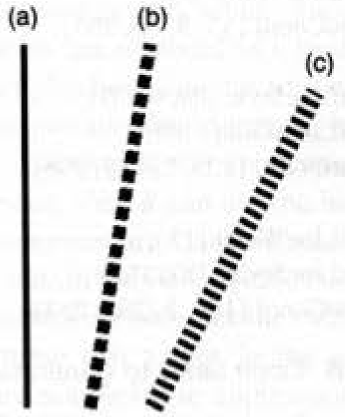


Fig. 2.7 Lines of various widths and styles.

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."

```
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, 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

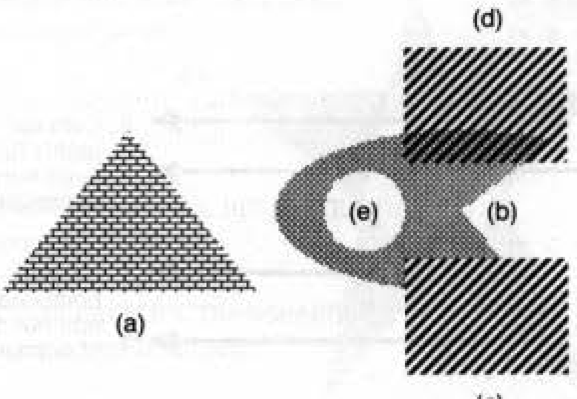


Fig. 2.9 Filled primitives. (a-c) Bitmap pattern opaque. (d) Bitmap pattern transparent. (e) Solid.

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

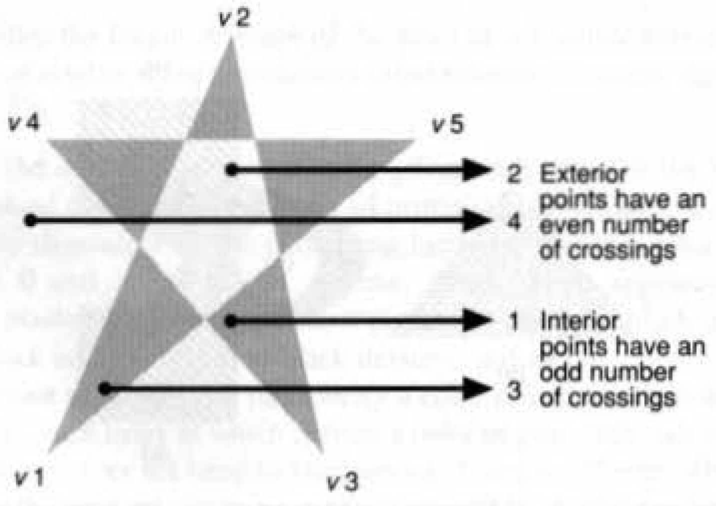

v2
v4
v5
2 Exterior points have an
4 even number of crossings
1 Interior points have an
3 odd number of crossings
v1
v3

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

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.

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

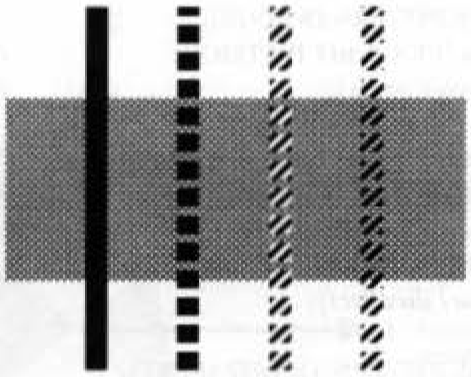


Fig. 2.12 Interaction between pen style and line style. (a) Continuous solid. (b) Dashed solid. (c) Dashed bitmap pattern opaque. (d) Dashed bitmap pattern transparent.

```
/* 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).

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*[5]) 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.

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

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

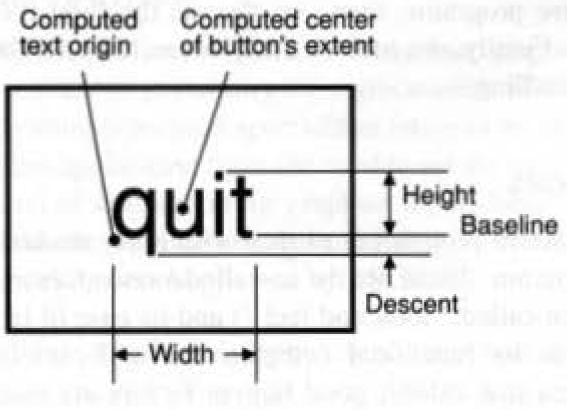


Fig. 2.14 Dimensions of text centered within a rectangular button and points computed from these dimensions for centering purposes.

```
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:

- 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

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

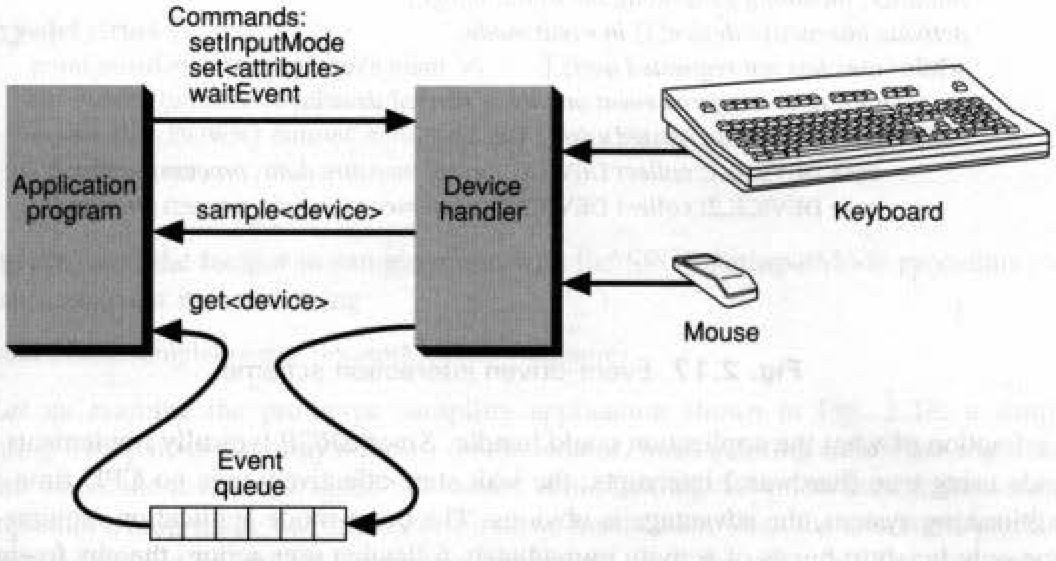


Fig. 2.16 Sampling versus event-handling using the event queue.

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

```
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.

```
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.

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 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.

```
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

```
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.)

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.

```
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

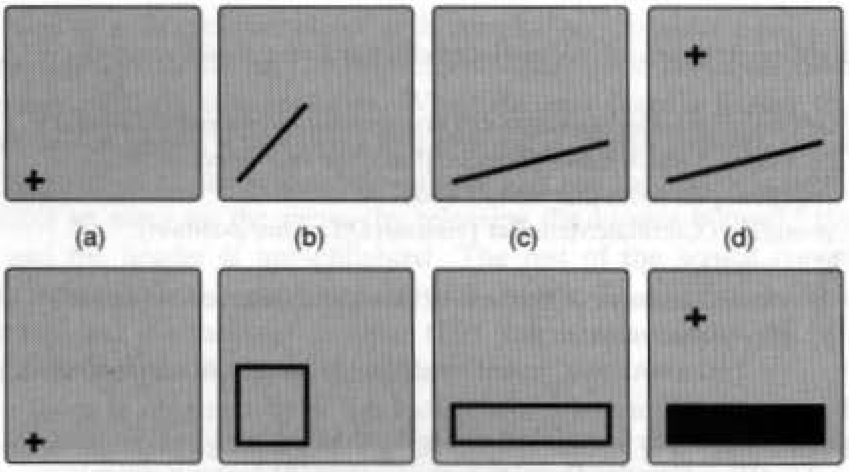


Fig. 2.23 Rubber-echo scenarios.

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 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).

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.

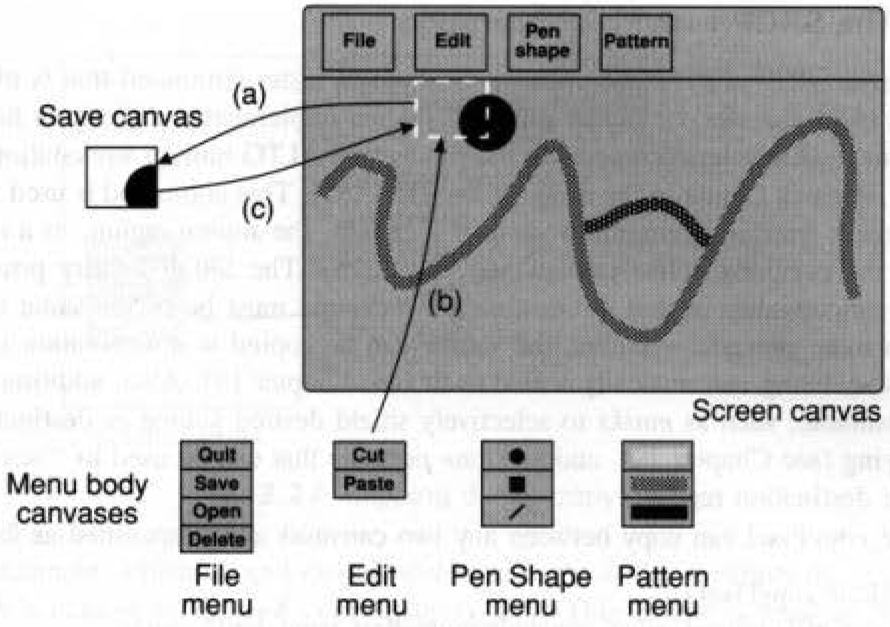


Fig. 2.25 Saving and restoring area covered by menu body.

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.

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.

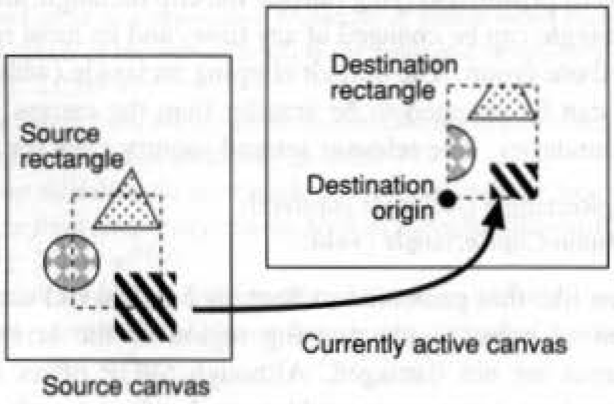


Fig. 2.26 SRGP_copyPixel.

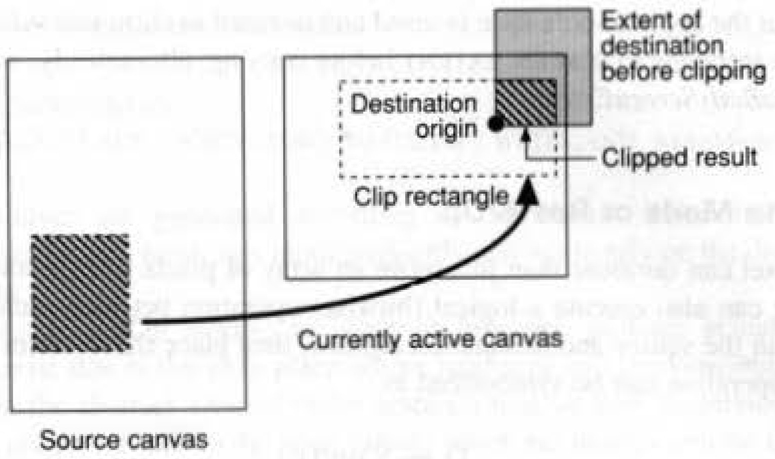


Fig. 2.27 Clipping during copyPixel.

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.

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.

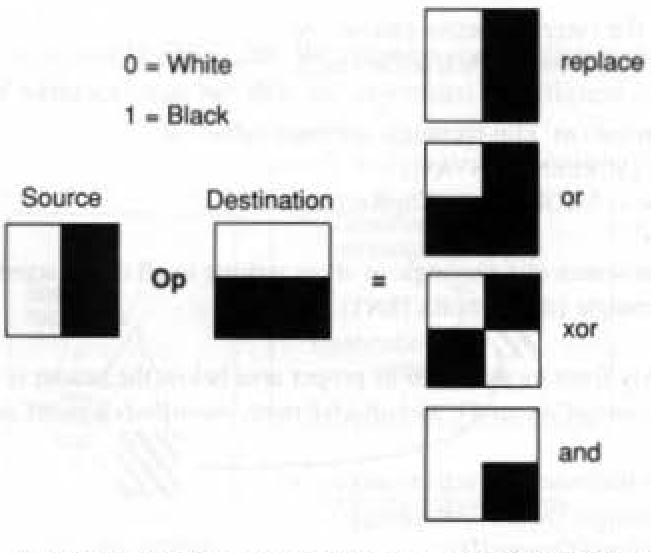


Fig. 2.29 Write modes for combining source and destination pixels.

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 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 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

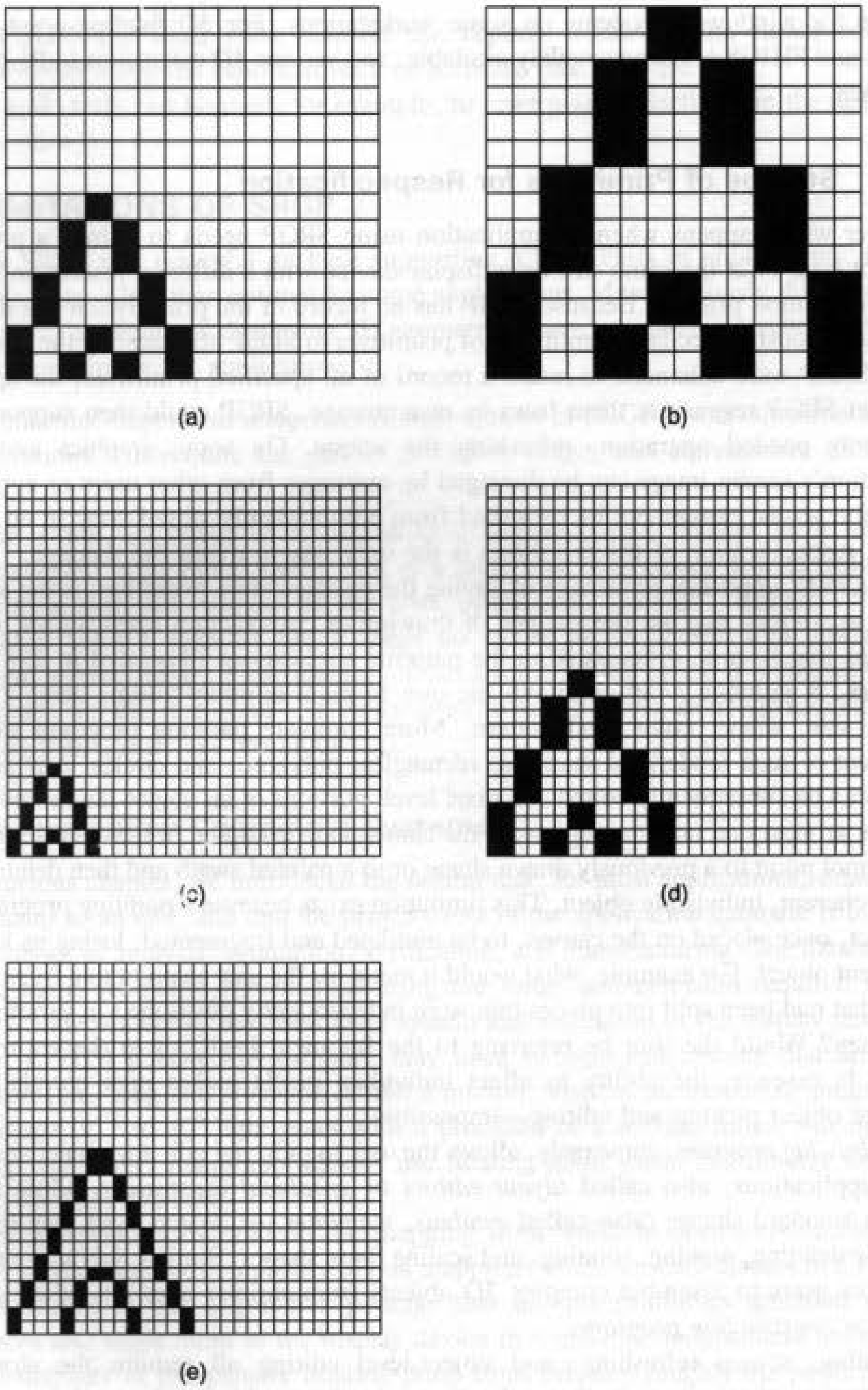


Fig. 2.30 The effects of pixel replication. (a) Original image at screen resolution. (b) Zoomed (2×) image at screen resolution. (c) Original image printed on device with twice the screen's resolution. (d) Zoomed image on same device as (c), using pixel replication to maintain image size. (e) Original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

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 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?

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.

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.

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-

TEXAS INSTRUMENTS EX. 1009 - 90/1253

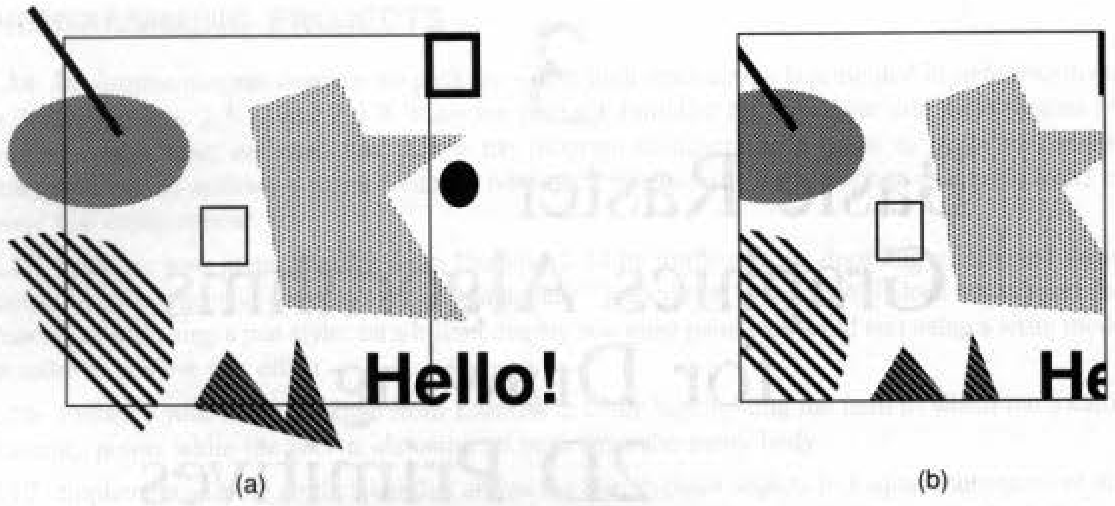


Fig. 3.1 Clipping SRGP primitives to a rectangular clip region. (a) Primitives and clipping rectangle. (b) Clipped results.

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

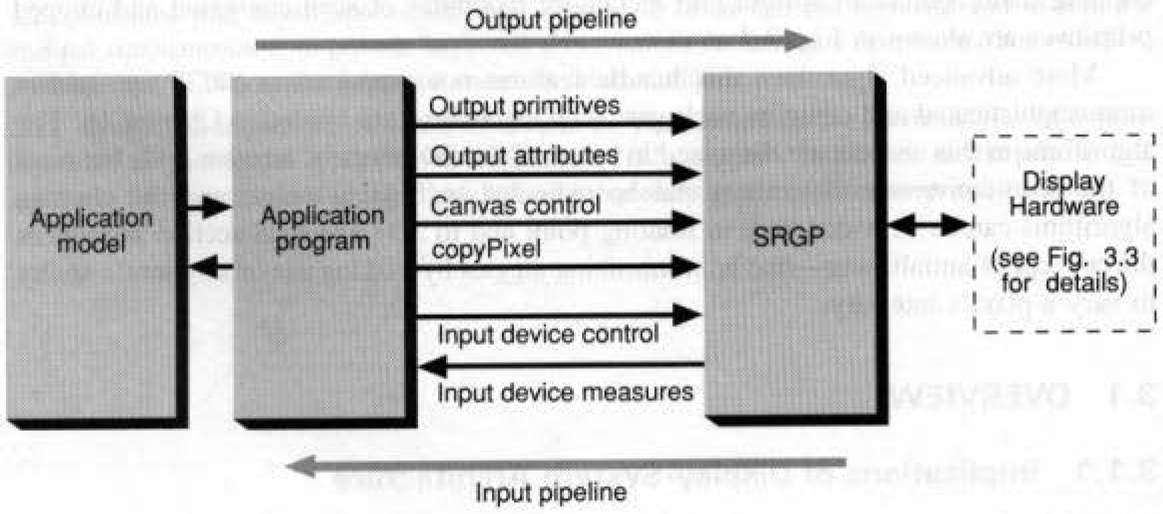


Fig. 3.2 SRGP as intermediary between the application program and the graphics system, providing output and input pipelines.

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.

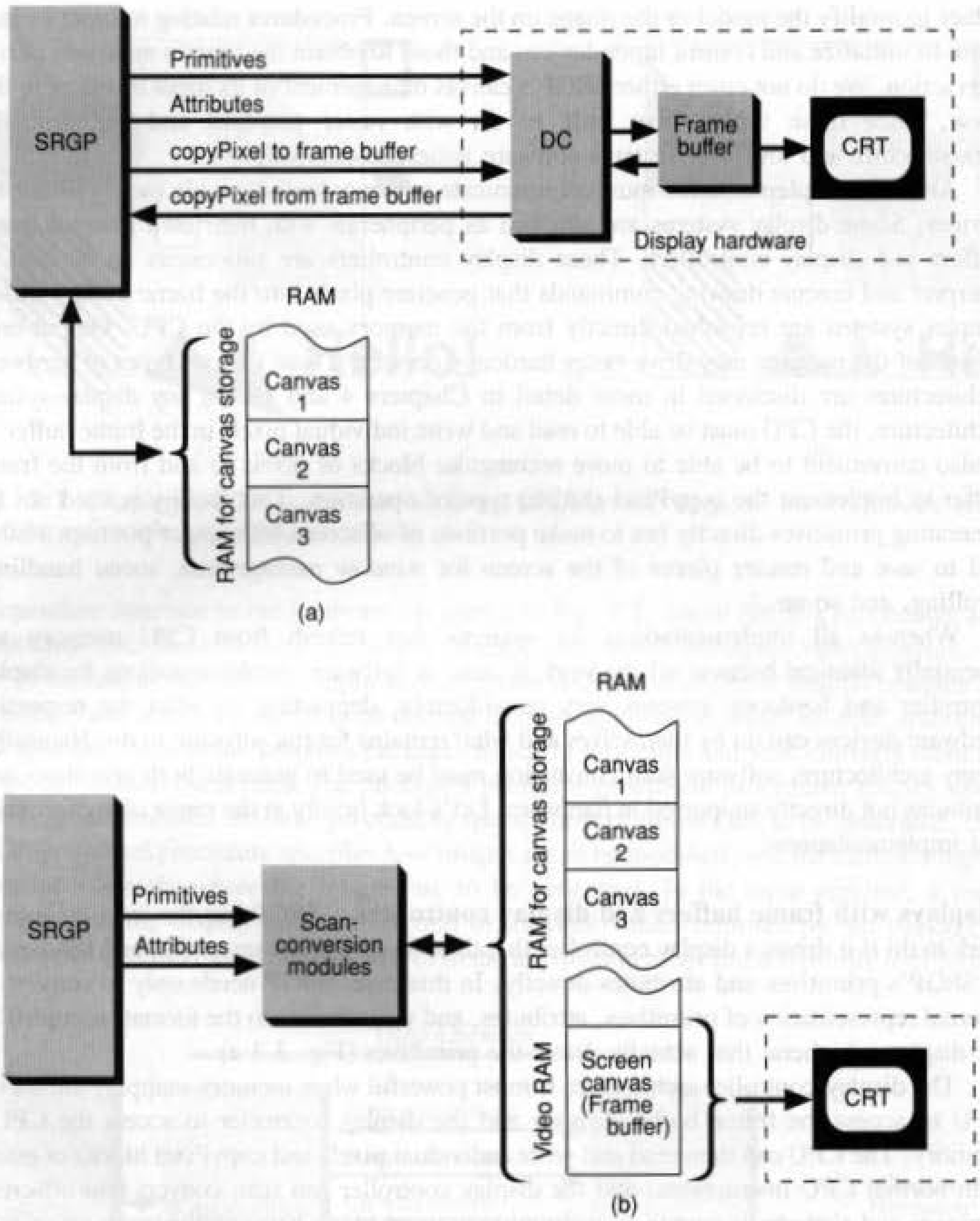


Fig. 3.3 SRGP driving two types of display systems. (a) Display peripheral with display controller and frame buffer. (b) No display controller, memory-shared frame buffer.

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 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.

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

3.2 SCAN CONVERTING LINES

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

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

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

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

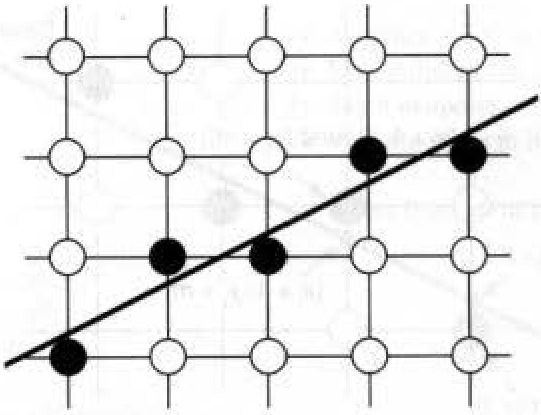


Fig. 3.4 A scan-converted line showing intensified pixels as black circles.

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

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

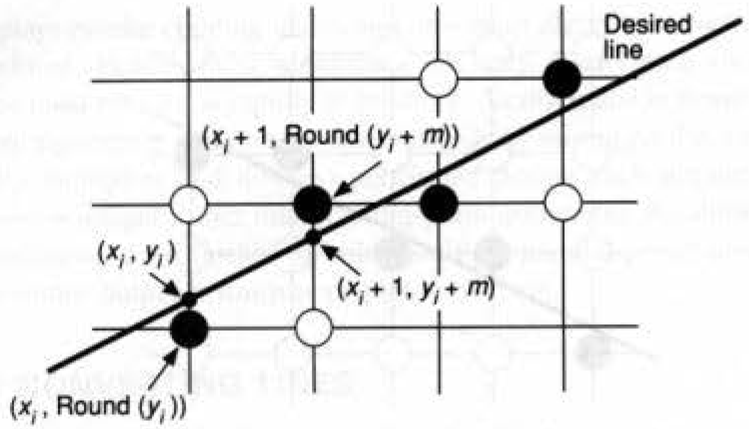

Desired line
$(x_i + 1, \text{Round}(y_i + m))$
(x_i, y_i)
$(x_i + 1, y_i + m)$
$(x_i, \text{Round}(y_i))$

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 $-1 \le m \le 1$, 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.

```
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

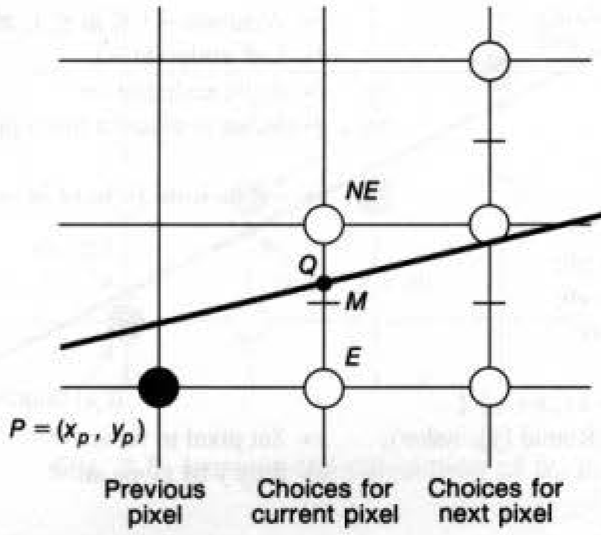


Fig. 3.7 The pixel grid for the midpoint line algorithm, showing the midpoint *M*, and the *E* and *NE* pixels to choose between.

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 $y_1 > y_0$.

incremented by one step in the x direction. Then,

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

but

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

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

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,

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

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

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

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 $(x_0 + 1, y_0 + \tfrac{1}{2})$, 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.

```
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.

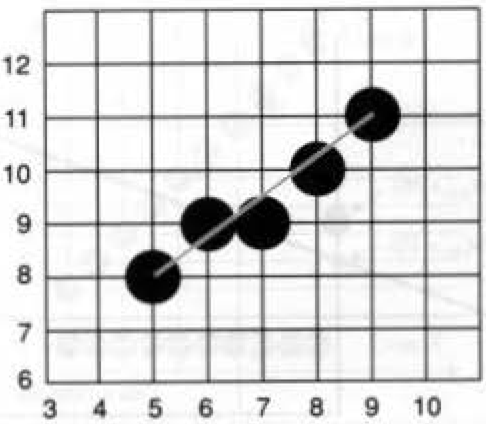


Fig. 3.9 The midpoint line from point (5, 8) to point (9, 11).

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 $d = 0$.

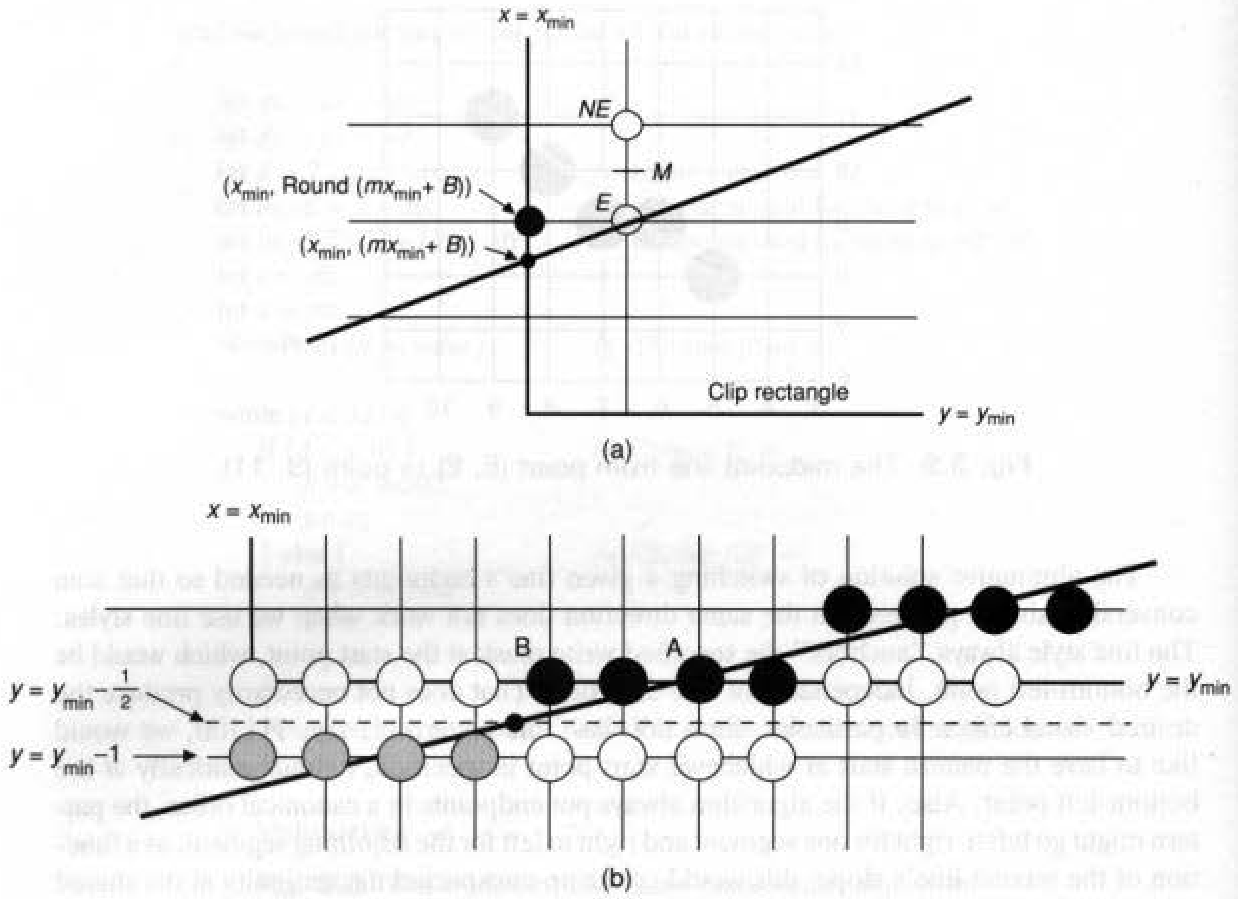


Fig. 3.10 Starting the line at a clip boundary. (a) Intersection with a vertical edge. (b) Intersection with a horizontal edge (gray pixels are on the line but are outside the clip rectangle).

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

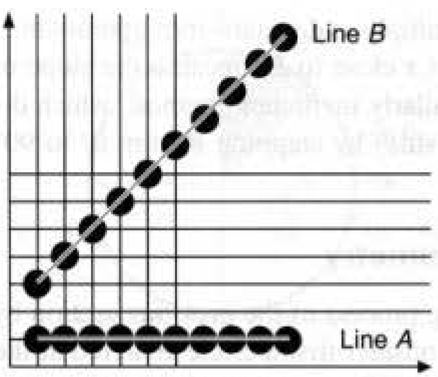


Fig. 3.11 Varying intensity of raster lines as a function of slope.

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 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.

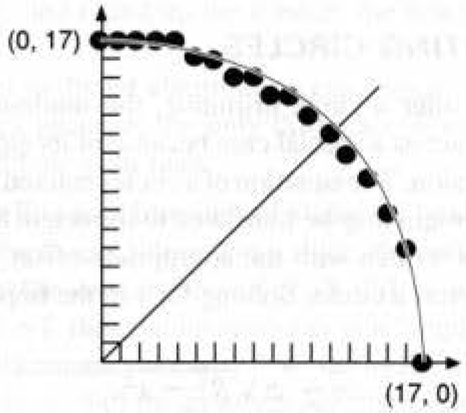


Fig. 3.12 A quarter circle generated with unit steps in x, and with y calculated and then rounded. Unique values of y for each x produce gaps.

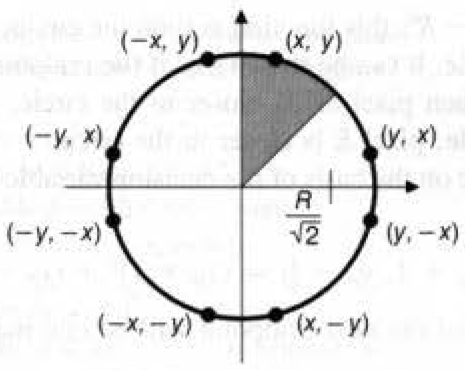


Fig. 3.13 Eight symmetrical points on a circle.

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).

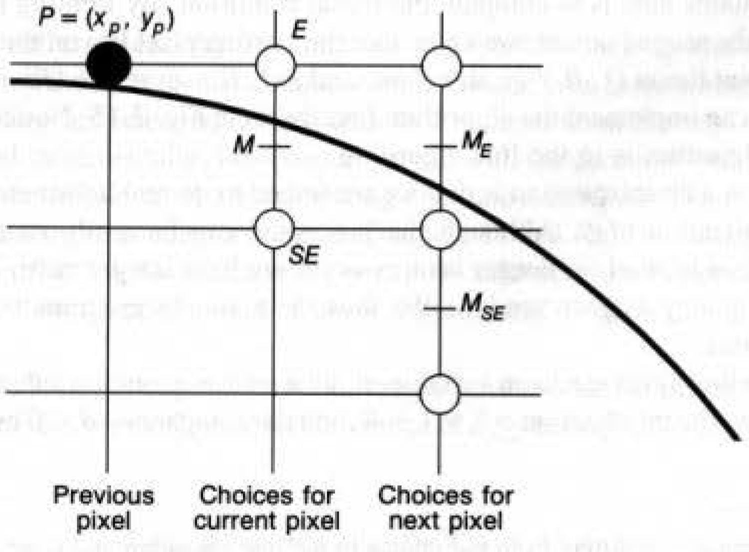


Fig. 3.14 The pixel grid for the midpoint circle algorithm showing M and the pixels E and SE to choose between.

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

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

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

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

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

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

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

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

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

Recall that, in the linear case, 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 $(1, R - \tfrac{1}{2})$, therefore, and $F(1, R - \tfrac{1}{2}) = 1 + (R^2 - R + \tfrac{1}{4}) - R^2 = \tfrac{5}{4} - R$. 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 $h = d - \tfrac{1}{4}$, and we substitute $h + \tfrac{1}{4}$ for d in the code. Now, the intialization is $h = 1 - R$, and the comparison $d < 0$ becomes $h < -\tfrac{1}{4}$.

[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.

```
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 $\Delta_{E_{new}} - \Delta_{E_{old}} = 2$.

```
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.

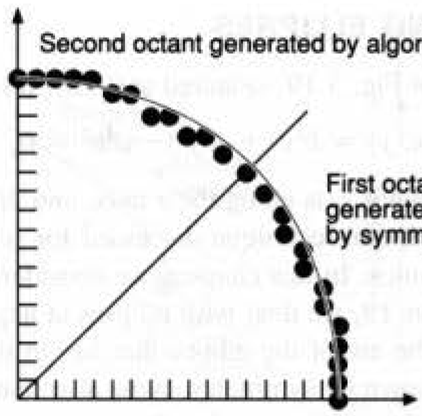

Second octant generated by algorithm
First octant generated by symmetry

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.

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 $\mathbf{i}$ and $\mathbf{j}$ components of the gradient are of equal magnitude. The $\mathbf{j}$ component of the gradient is larger than the $\mathbf{i}$ component in region 1, and vice versa in region 2. Thus, if at the next midpoint, $a^2(y_P - \frac{1}{2}) \le b^2(x_P + 1)$, we switch from region 1 to region 2.

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

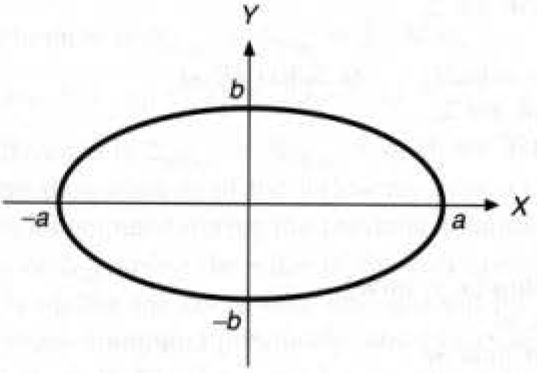


Fig. 3.19 Standard ellipse centered at the origin.

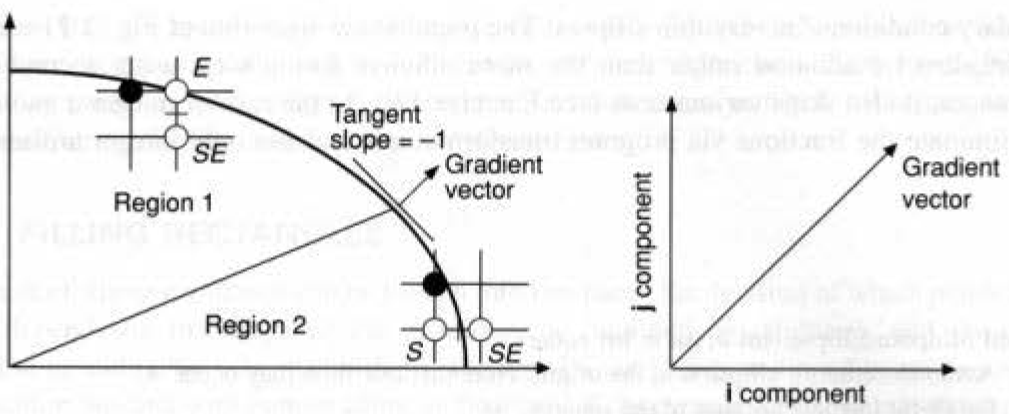


Fig. 3.20 Two regions of the ellipse defined by the 45° tangent.

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

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

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

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

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

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

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

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

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

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 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.

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 y_min to y_max of the rectangle)     /* By scan line */
    for (x from x_min to x_max )                 /* 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 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

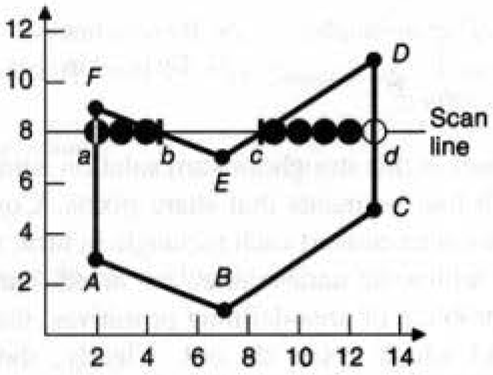


Fig. 3.22 Polygon and scan line 8.

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:

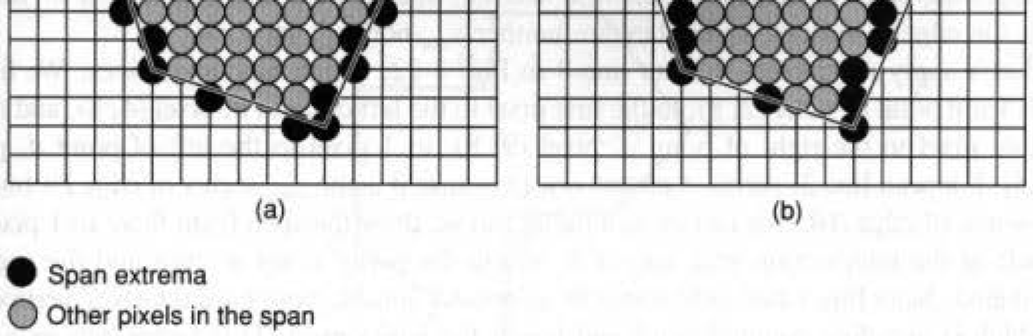


Fig. 3.23 Spans for a polygon. Extrema shown in black, interior pixels in gray. (a) Extrema computed by midpoint algorithm. (b) Extrema interior to polygon.

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

3.6.1 Horizontal Edges

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

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

3.6.2 Slivers

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

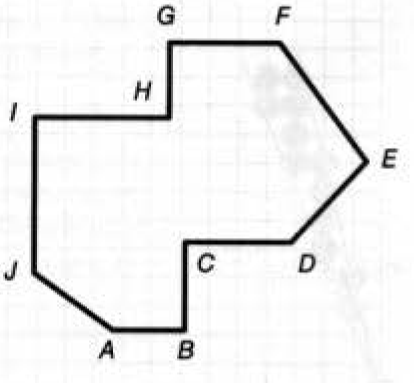


Fig. 3.24 Horizontal edges in a polygon.

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

3.6.3 Edge Coherence and the Scan-Line Algorithm

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

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

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

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

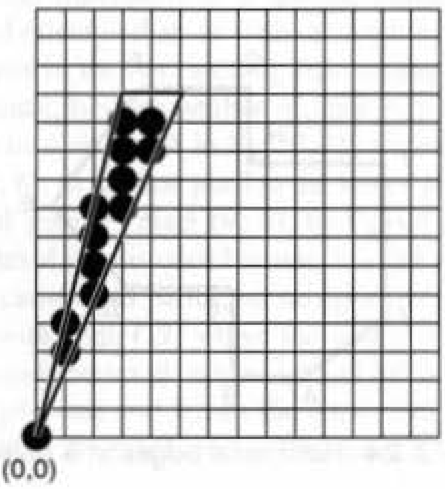


Fig. 3.25 Scan converting slivers of polygons.

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

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

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

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

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

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

Figure 3.26 Scan converting left edge of a polygon.

TEXAS INSTRUMENTS EX. 1009 - 120/1253

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

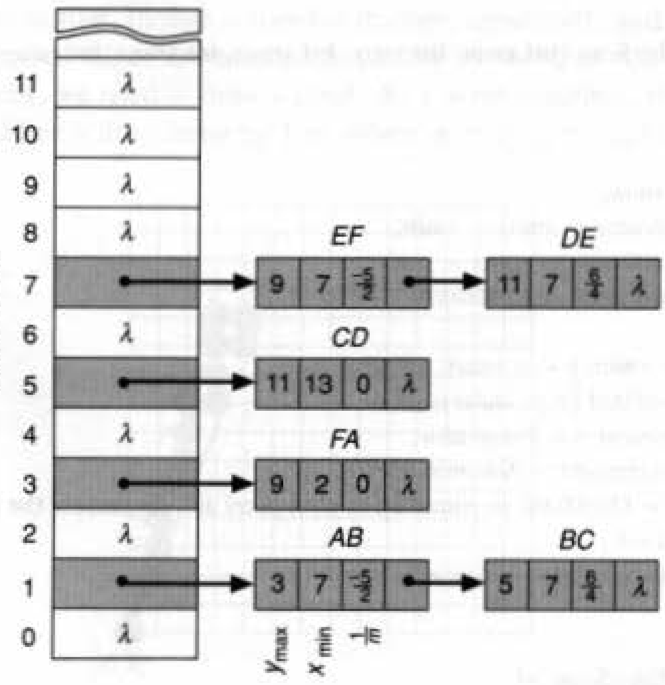


Fig. 3.27 Bucket-sorted edge table for polygon of Fig. 3.22.

TEXAS INSTRUMENTS EX. 1009 - 121/1253

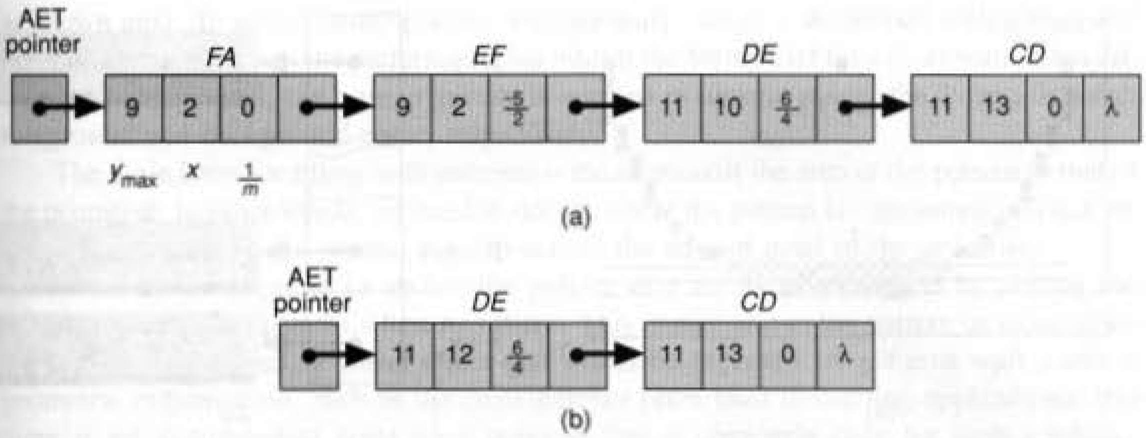


Fig. 3.28 Active-edge table for polygon of Fig. 3.22. (a) Scan line 9. (b) Scan line 10. (Note *DE*'s *x* coordinate in (b) has been rounded up for that left edge.)

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

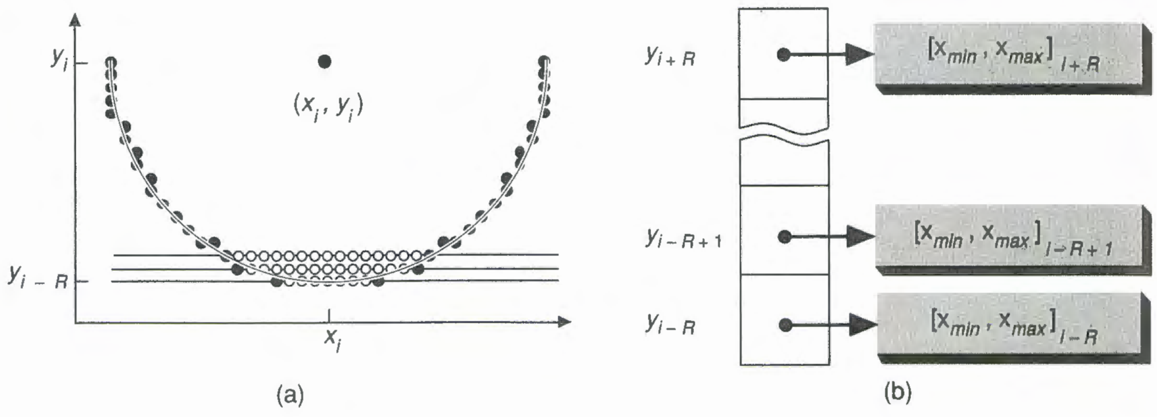

y_i
(x_i, y_i)
y_{i-R}
x_i
(a)
y_{i+R}
$[x_{min}, x_{max}]_{i+R}$
y_{i-R+1}
$[x_{min}, x_{max}]_{i-R+1}$
y_{i-R}
$[x_{min}, x_{max}]_{i-R}$
(b)

Fig. 3.29 Filling a circle with spans. (a) Three spans. (b) Span table. Each span is stored with its extrema.

unfilled primitive, we can take advantage of symmetry to scan convert only one arc, being careful of region changes, especially for ellipses. Each iteration generates either a new pixel on the same scan line, thereby potentially adjusting the span extrema, or a pixel on the next scan line, starting the next span (see Fig. 3.29). To determine whether a pixel P lies inside the region, we simply check the sign of the function $F(P)$ and choose the next pixel over if the sign is positive. Clearly, we do not want to evaluate the function directly, any more than we wanted to evaluate the function $F(M)$ at the midpoint directly; we can derive the value of the function from that of the decision variable.

Since we know that a scan line crosses the boundary only twice, we do not need any equivalent of an edge table, and we maintain only a current span. As we did for polygons, we can either make a list of such spans for each scan line intersecting the primitive and then fill them after they have all been computed, or we can fill each one as it is completed—for example, as soon as the y value is incremented.

The special case of filled wedges should be mentioned. The first problem is to calculate the intersection of the rays that define the starting and ending angles with the boundary, and to use those to set the starting and ending values of the decision variable. For circles we can do this calculation by converting from rectangular degrees to circular degrees, then using (Round(R cosθ), Round(R sinθ)) in the midpoint formula. The perimeter of the region to scan convert, then, consists of the two rays and the boundary arc between them. Depending on the angles, a scan line may start or end on either a ray or the boundary arc, and the corresponding incremental algorithms, modified to select only interior pixels, must be applied (see Exercise 3.19).

3.8 PATTERN FILLING

In the previous sections, we filled the interiors of SRGP's area-defining primitives with a solid color by passing the color in the *value* field of the WritePixel procedure. Here, we consider filling with a pattern, which we do by adding extra control to the part of the scan-conversion algorithm that actually writes each pixel. For pixmap patterns, this control causes the color value to be picked up from the appropriate position in the pixmap pattern, 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.

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.

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

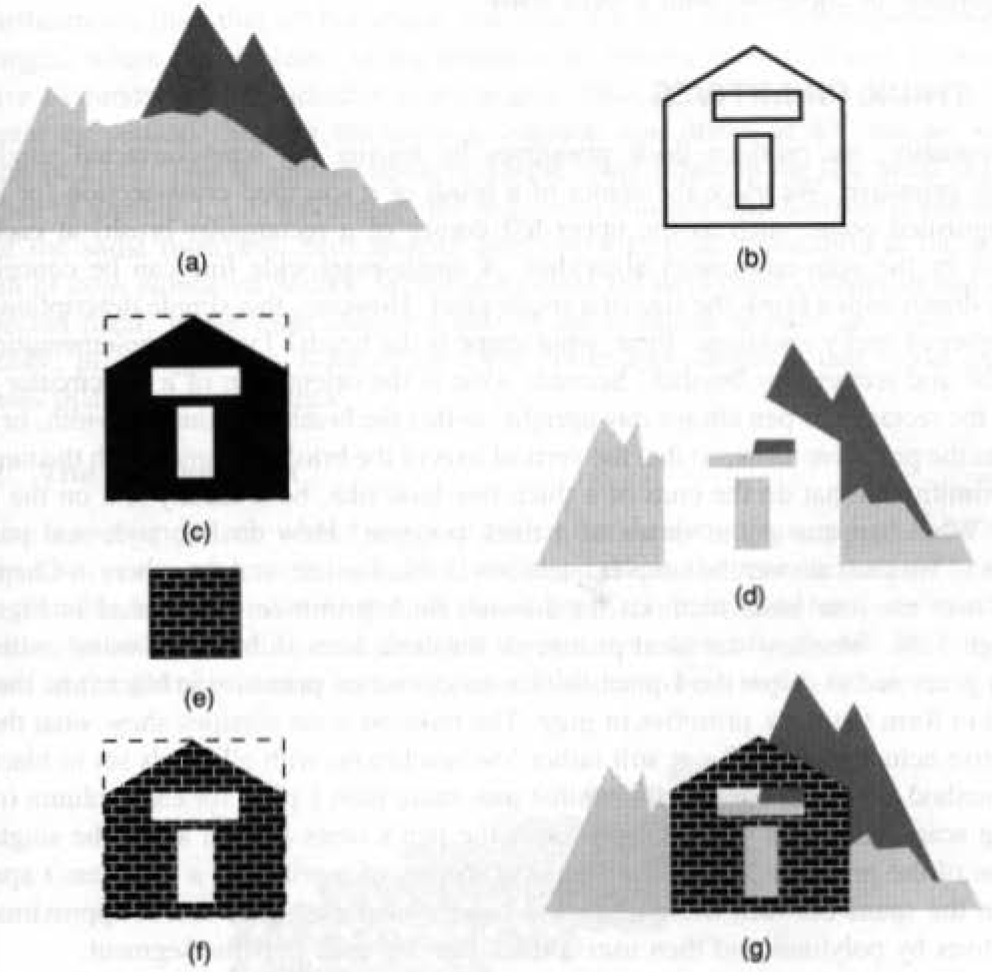


Fig. 3.30 Writing a patterned object in opaque mode with two transparent writes. (a) Mountain scene. (b) Outline of house icon. (c) Bitmap for solid version of house icon. (d) Clearing the scene by writing background. (e) Brick pattern. (f) Brick pattern applied to house icon. (g) Writing the screen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

TEXAS INSTRUMENTS EX. 1009 - 126/1253

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 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

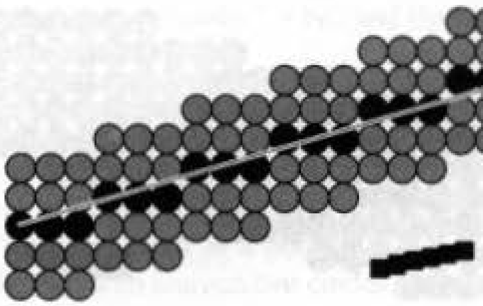


Fig. 3.31 Thick line drawn by column replication.

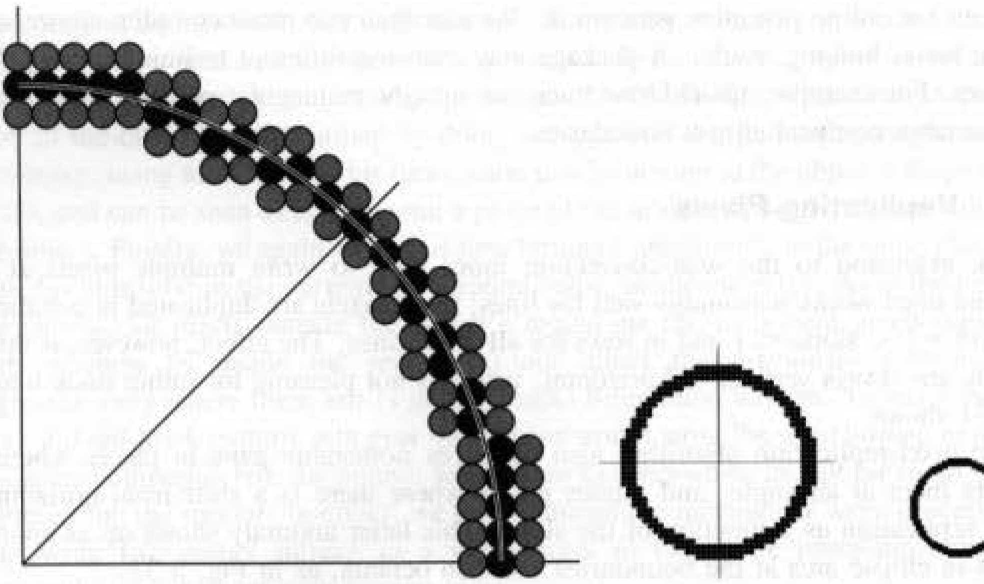


Fig. 3.32 Thick circle drawn by column replication.

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

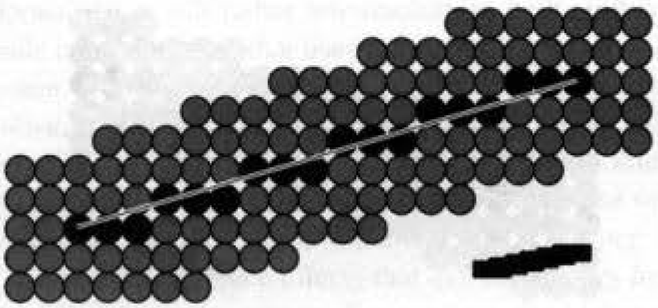


Fig. 3.33 Thick line drawn by tracing a rectangular pen.

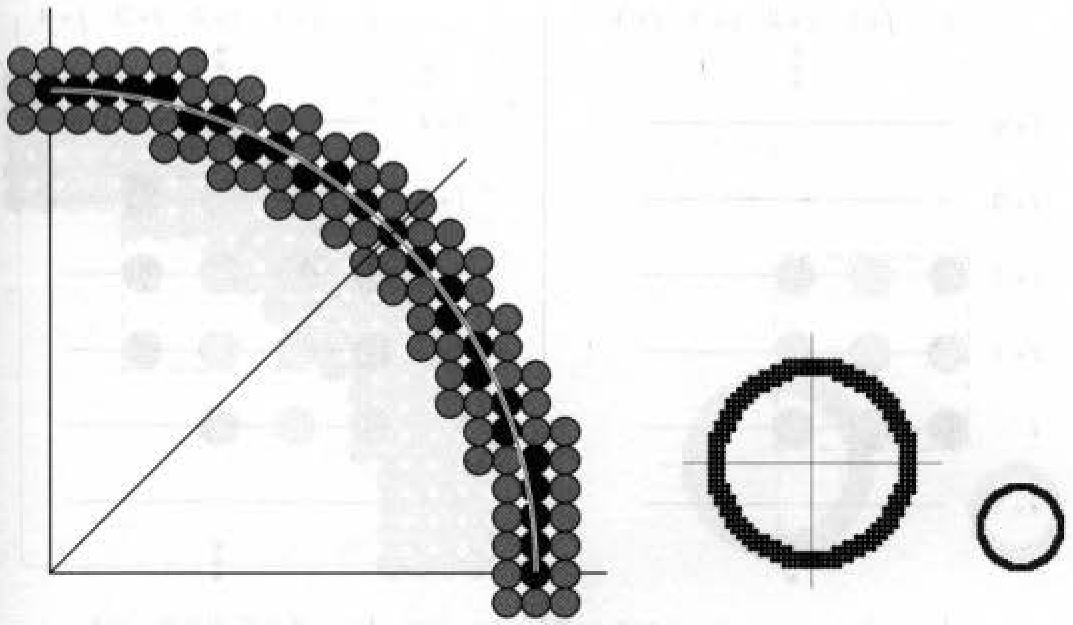


Fig. 3.34 Thick circle drawn by tracing a rectangular pen.

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

3.9.3 Filling Areas Between Boundaries

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

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

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

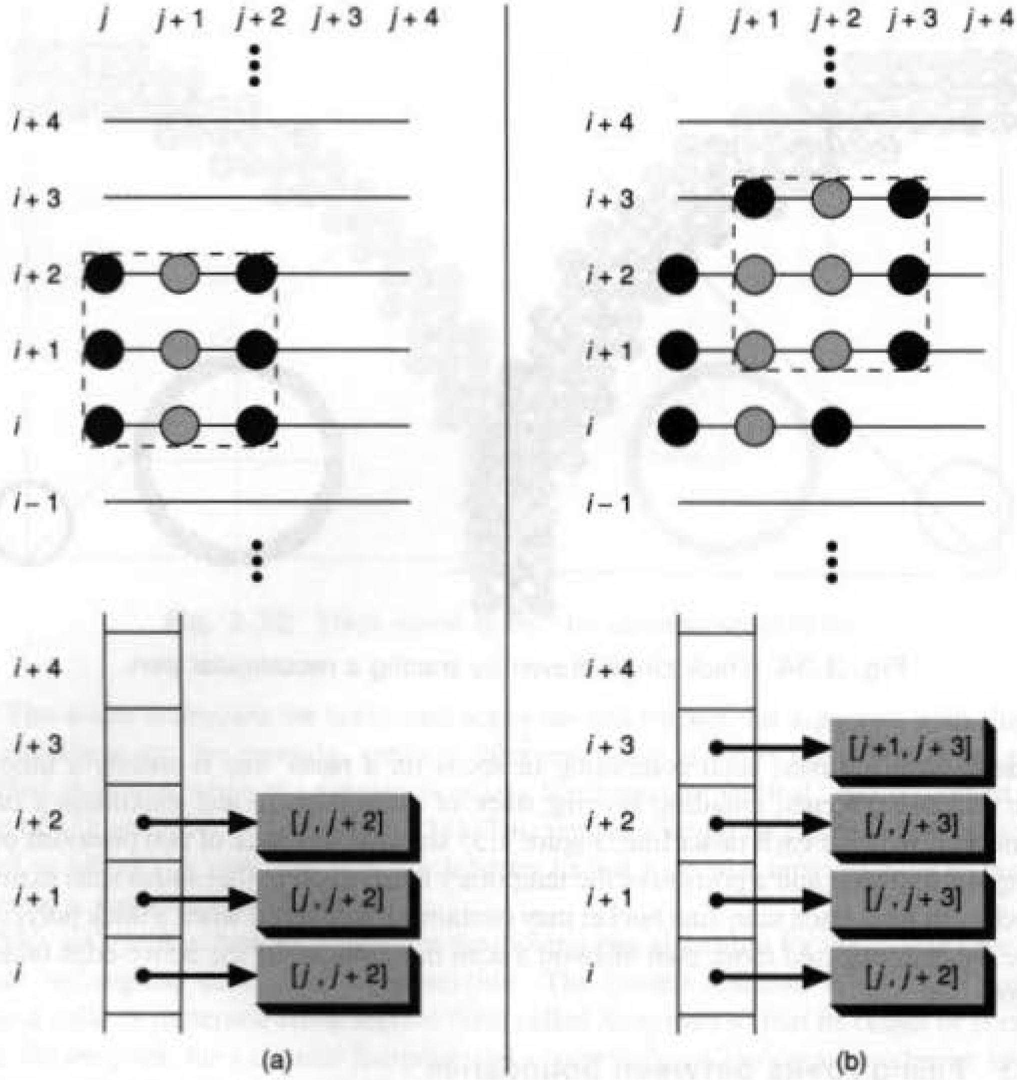


Fig. 3.35 Recording spans of the rectangular pen: (a) footprint at $x = j + 1$; (b) $x = j + 2$.

For ellipses, the situation is not nearly so simple. It is a classic result in differential geometry that the curves formed by moving a distance $t/2$ perpendicular to an ellipse are not confocal ellipses, but are described by eighth-order equations [SALM96].[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.

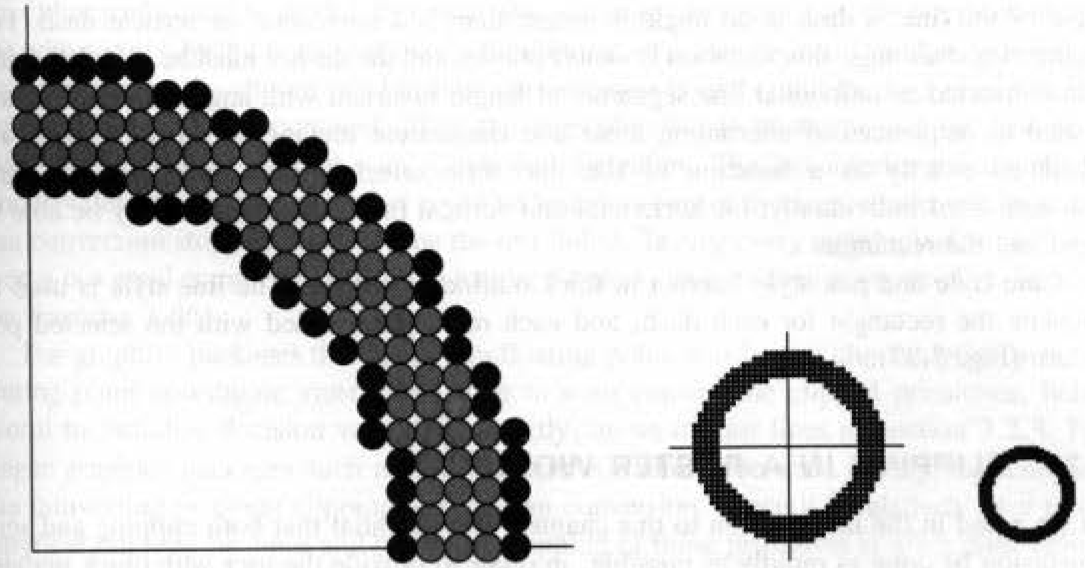


Fig. 3.36 Thick circle drawn by filling between concentric circles.

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 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.

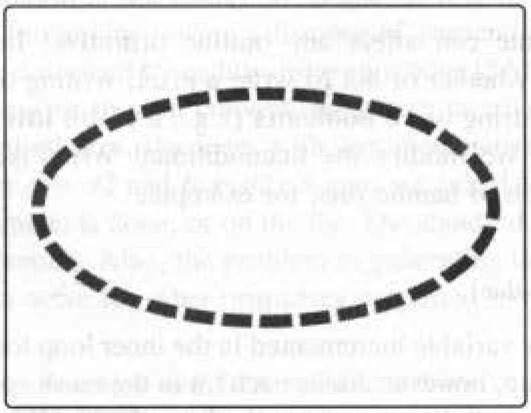


Fig. 3.37 Combining pen pattern and line style.

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 *i*th 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.

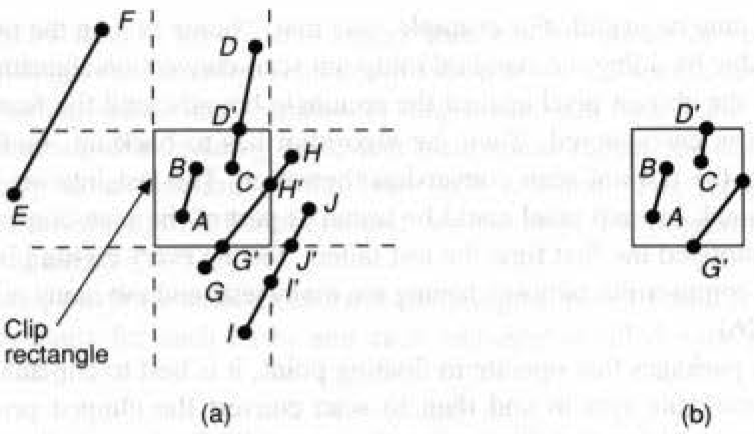


Fig. 3.38 Cases for clipping lines.

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:

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

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 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:

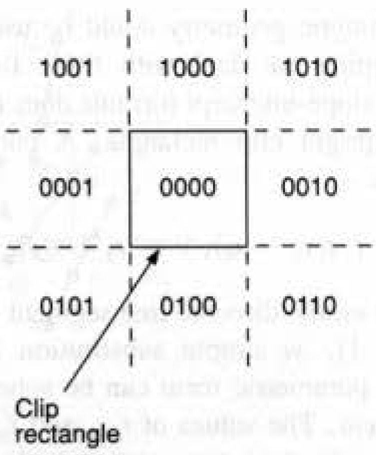


Fig. 3.39 Region outcodes.

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

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

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

The algorithm works as follows. We compute the outcodes of both endpoints and check for trivial acceptance and rejection. If neither test is 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

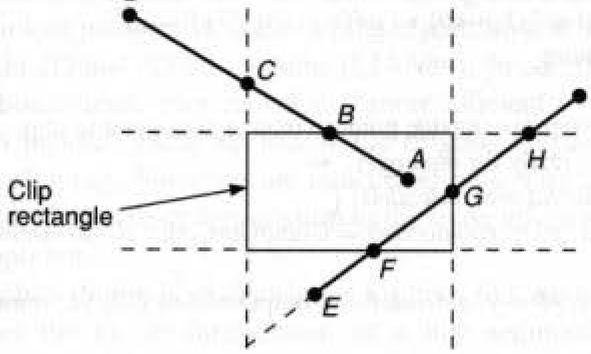


Fig. 3.40 Illustration of Cohen–Sutherland line clipping.

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

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

Fig. 3.41 *(Cont.)*.

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

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

Fig. 3.41 Cohen–Sutherland line-clipping algorithm.

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

3.12.4 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 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.

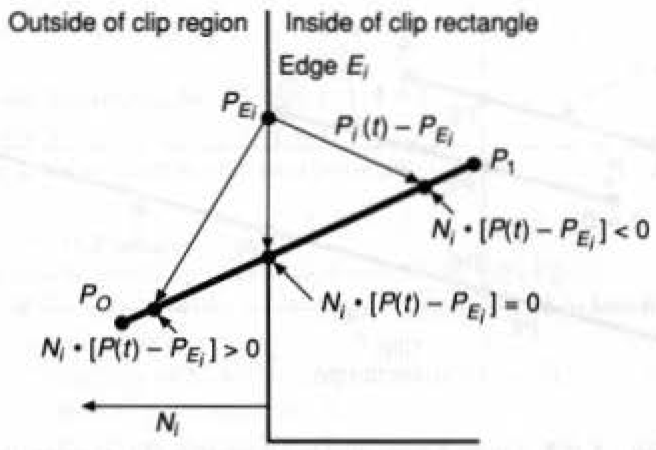


Fig. 3.42 Dot products for three points outside, inside, and on the boundary of the clip region.

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:

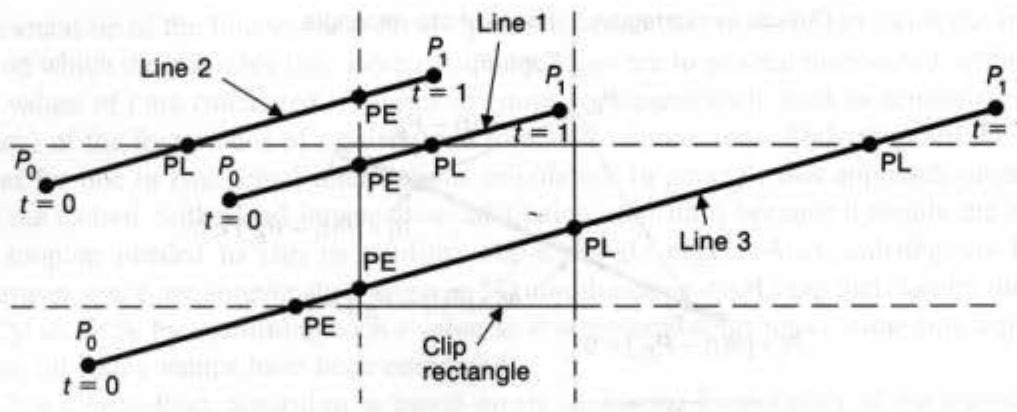


Fig. 3.43 Lines lying diagonal to the clip rectangle.

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

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.

```
precalculate $N_i$ and select a $P_{E_i}$ for each edge;

for (each line segment to be clipped) {
    if ($P_1 == P_0$)
        line is degenerate so clip as a point;
    else {
        $t_E = 0$; $t_L = 1$;
        for (each candidate intersection with a clip edge) {
            if ($N_i \bullet D$ != 0) {      /* Ignore edges parallel to line for now */
                calculate $t$;
                use sign of $N_i \bullet D$ 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.

TEXAS INSTRUMENTS EX. 1009 - 144/1253

```
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 −(Ni • 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 Ni • (P0 − PEi) */
/* for a particular edge/line combination, which reduces to directed horizontal */
/* and vertical distances from P0 to an edge; its sign determines visibility */
/* of P0 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.)*.

```
{
    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 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

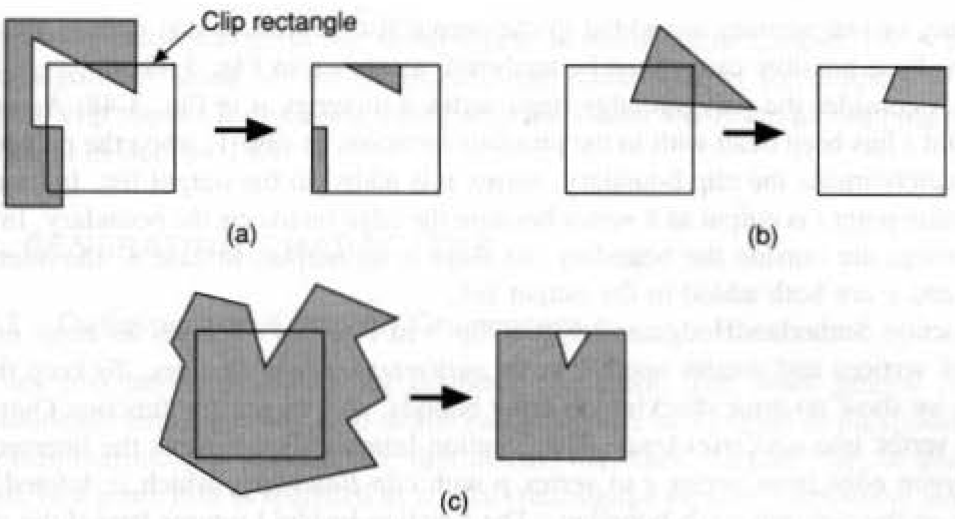


Fig. 3.46 Examples of polygon clipping. (a) Multiple components. (b) Simple convex case. (c) Concave case with many exterior edges.

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,

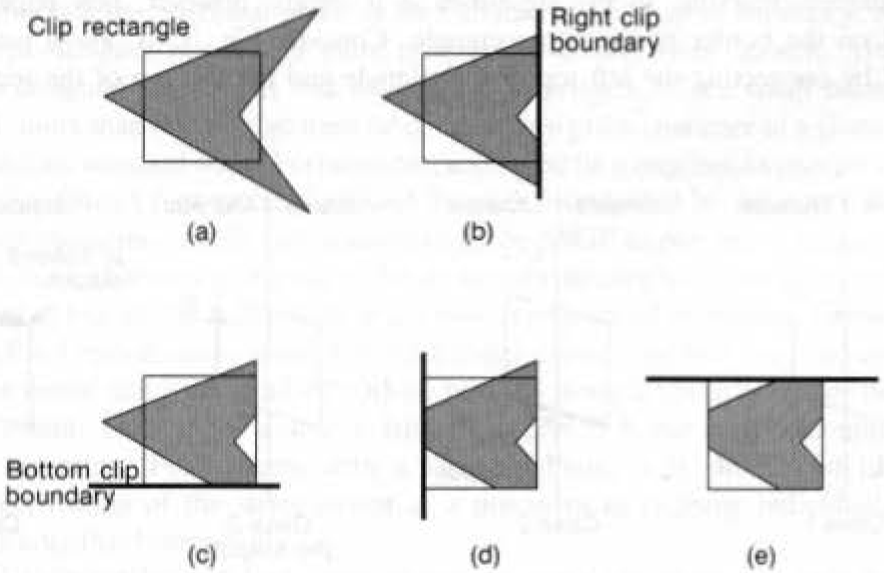


Fig. 3.47 Polygon clipping, edge by edge. (a) Before clipping. (b) Clip on right. (c) Clip on bottom. (d) Clip on left. (e) Clip on top; polygon is fully clipped.

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

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

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

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

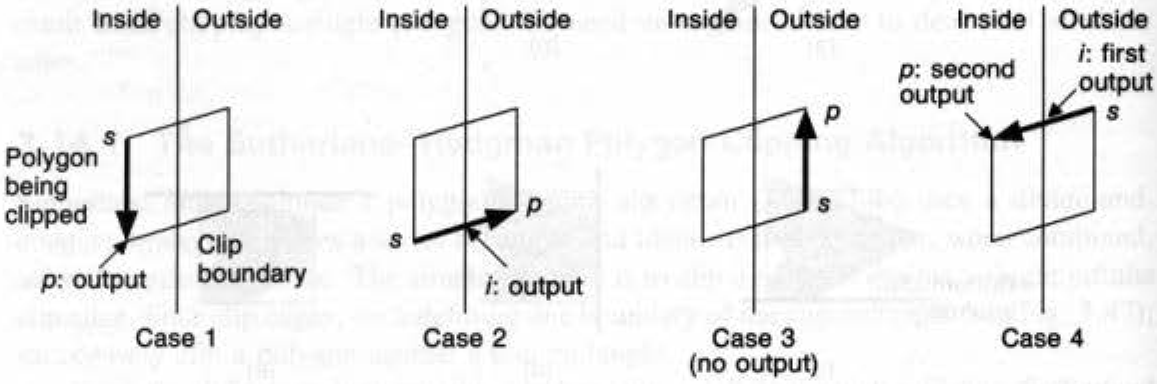


Fig. 3.48 Four cases of polygon clipping.

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

3.15 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

```
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.)*.

```
{
...
}

/* 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.

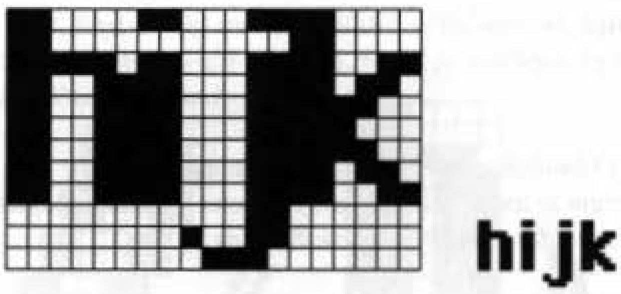


Fig. 3.50 Portion of an example of a font cache.

```
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.

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.

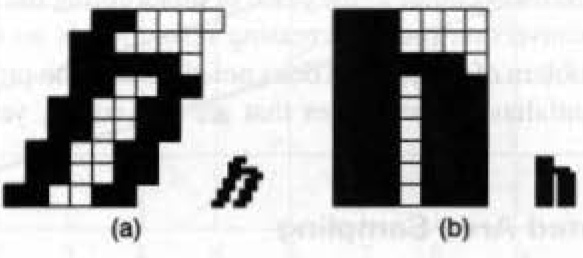


Fig. 3.53 Tricks for creating different faces for a font. (a) Italic. (b) Bold.

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

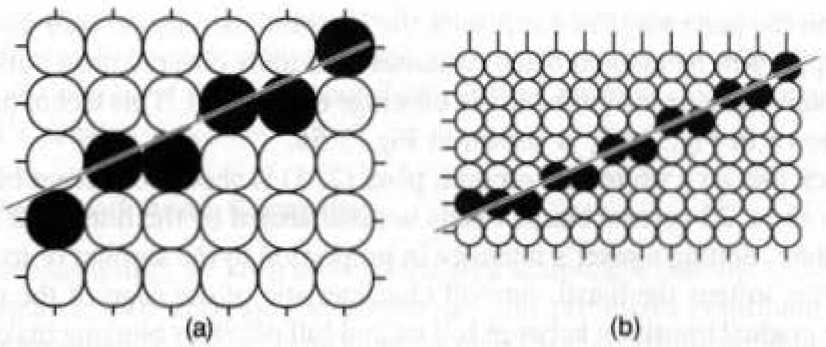


Fig. 3.54 (a) Standard midpoint line on a bilevel display. (b) Same line on a display that has twice the linear resolution.

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

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

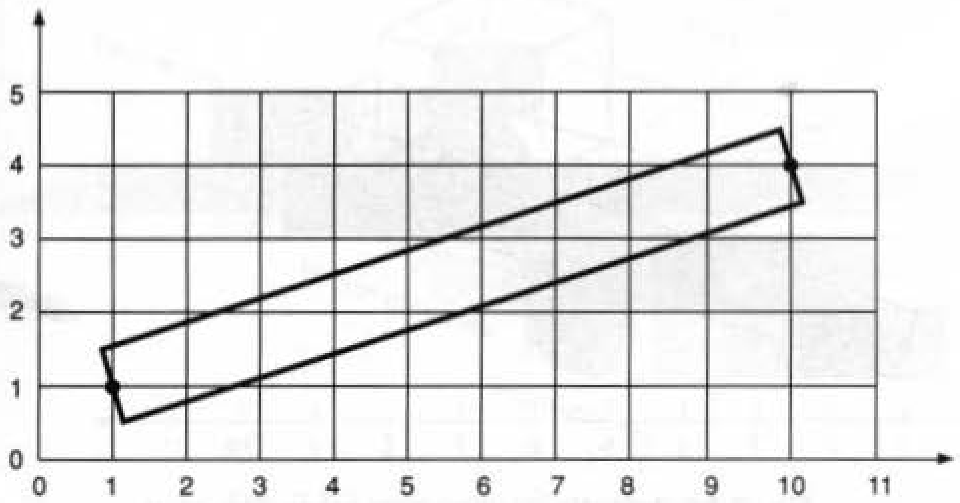


Figure 3.55 Line of nonzero width from point (1,1) to point (10,4).

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

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

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

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

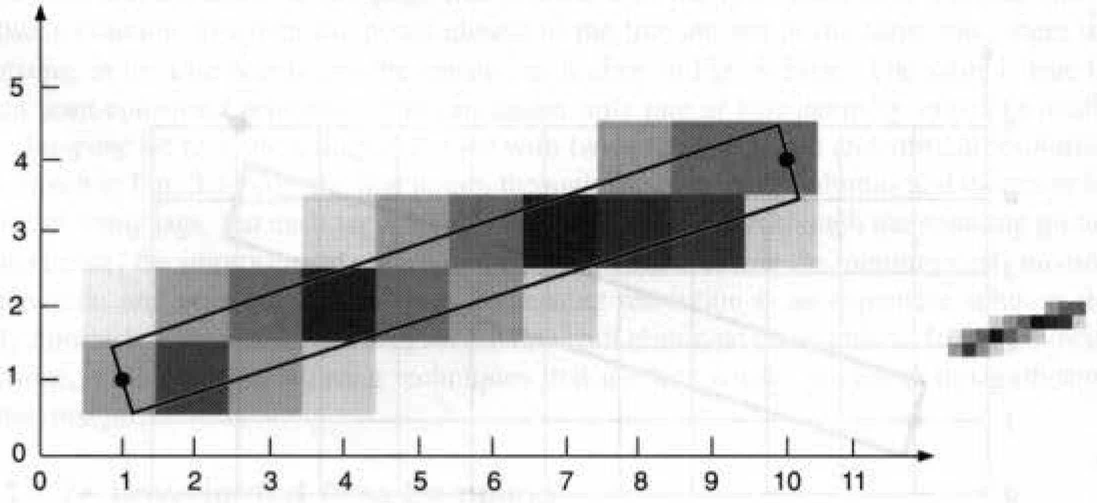


Fig. 3.56 Intensity proportional to area covered.

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

3.17.3 Weighted Area Sampling

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

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

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

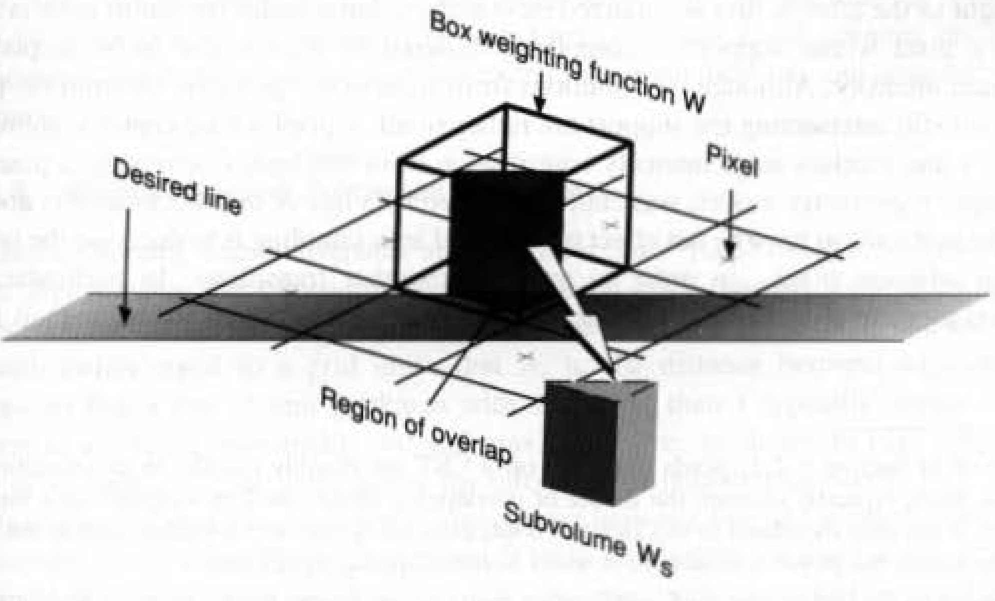


Fig. 3.57 Box filter for square pixel.

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

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

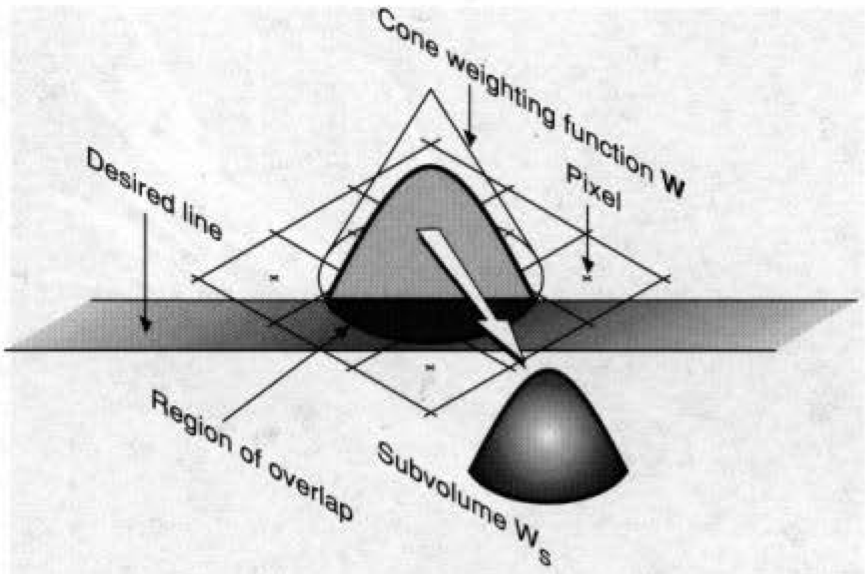


Fig. 3.58 Cone filter for circular pixel with diameter of two grid units.

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

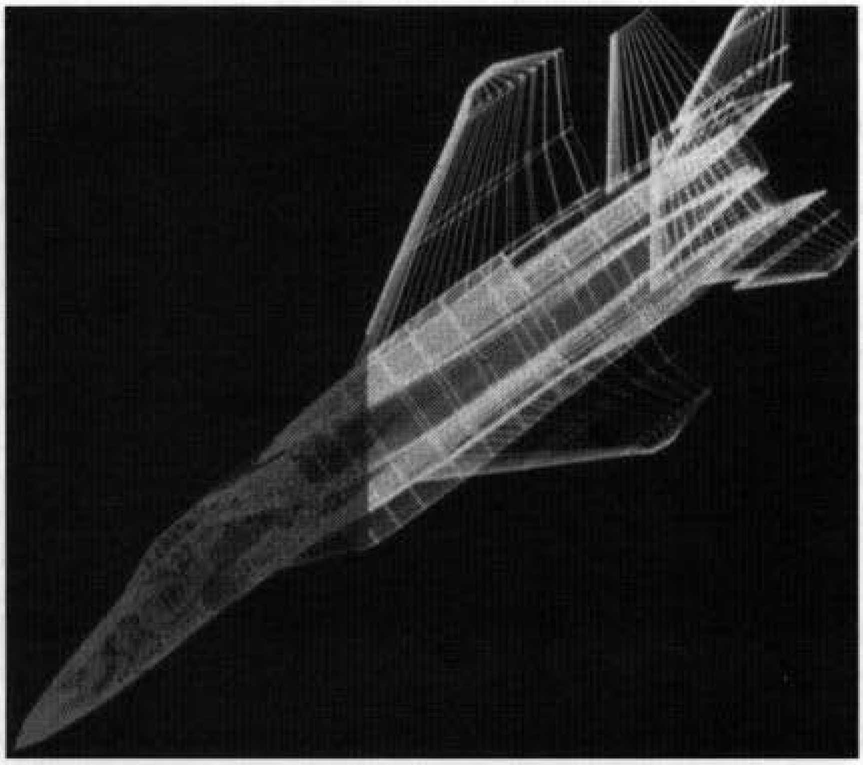


Fig. 3.59 Filtered line drawing. The left half is unfiltered; the right half is filtered. (Courtesy of Branko Gerovac, Digital Equipment Corporation.)

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

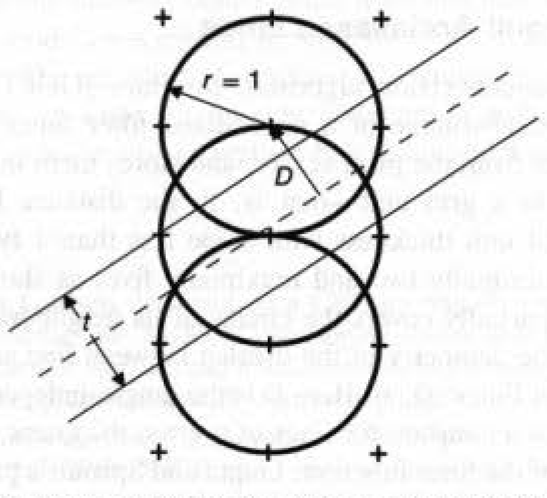


Fig. 3.60 One-unit-thick line intersects 3-pixel supports.

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

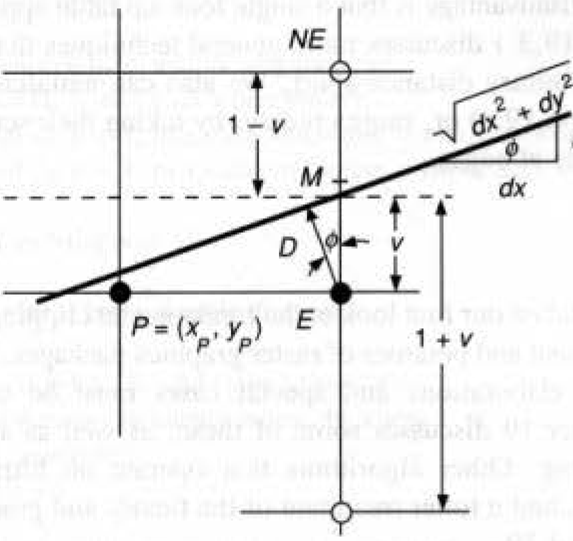


Fig. 3.61 Calculating distances to line in midpoint algorithm.

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.

```
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.

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.

3.13 Write a scan-conversion procedure for solidly filled upright rectangles at arbitrary screen positions that writes a bilevel frame buffer efficiently, an entire word of pixels at a time.

3.14 Construct examples of pixels that are "missing" or written multiple times, using the rules of Section 3.6. Try to develop alternative, possibly more complex, rules that do not draw shared pixels on shared edges twice, yet do not cause pixels to be missing. Are these rules worth the added overhead?

3.15 Implement the pseudocode of Section 3.6 for polygon scan conversion, taking into account in the span bookkeeping of potential sliver polygons.

3.16 Develop scan-conversion algorithms for triangles and trapezoids that take advantage of the simple nature of these shapes. Such algorithms are common in hardware.

3.17 Investigate triangulation algorithms for decomposing an arbitrary, possibly concave or self-intersecting, polygon into a mesh of triangles whose vertices are shared. Does it help to restrict the polygon to being, at worse, concave without self-intersections or interior holes? (See also [PREP85].)

3.18 Extend the midpoint algorithm for scan converting circles (Fig. 3.16) to handle filled circles and circular wedges (for pie charts), using span tables.

3.19 Extend the midpoint algorithm for scan converting ellipses (Fig. 3.21) to handle filled elliptical wedges, using span tables.

3.20 Implement both absolute and relative anchor algorithms for polygon pattern filling, discussed in Section 3.9, and contrast them in terms of visual effect and computational efficiency.

3.21 Apply the technique of Fig. 3.30 for writing characters filled with patterns in opaque mode. Show how having a copyPixel with a write mask may be used to good advantage for this class of problems.

3.22 Implement a technique for drawing various symbols such as cursor icons represented by small bitmaps so that they can be seen regardless of the background on which they are written. Hint: Define a mask for each symbol that "encloses" the symbol—that is, that covers more pixels than the symbol—and that draws masks and symbols in separate passes.

3.23 Implement thick-line algorithms using the techniques listed in Section 3.9. Contrast their efficiency and the quality of the results they produced.

3.24 Extend the midpoint algorithm for scan converting circles (Fig. 3.16) to handle thick circles.

3.25 Implement a thick-line algorithm that accommodates line style as well as pen style and pattern.

3.26 Implement scissoring as part of scan converting lines and unfilled polygons, using the fast-scan-plus-backtracking technique of checking every *i*th pixel. Apply the technique to filled and thick lines and to filled polygons. For these primitives, contrast the efficiency of this type of on-the-fly clipping with that of analytical clipping.

3.27 Implement scissoring as part of scan converting unfilled and filled circles and ellipses. For these primitives, contrast the feasibility and efficiency of this type of on-the-fly clipping with that of analytical clipping.

3.28 Modify the Cohen–Sutherland line-clipping algorithm of Fig. 3.41 to avoid recalculation of slopes during successive passes.

3.29 Contrast the efficiency of the Sutherland–Cohen and Cyrus–Beck algorithms for several typical and atypical cases, using instruction counting. Are horizontal and vertical lines handled optimally?

3.30 Consider a convex polygon with *n* vertices being clipped against a clip rectangle. What is the maximum number of vertices in the resulting clipped polygon? What is the minimum number? Consider the same problem for a concave polygon. How many polygons might result? If a single polygon results, what is the largest number of vertices it might have?

3.31 Explain why the Sutherland–Hodgman polygon-clipping algorithm works for only convex clipping regions.

3.32 Devise a strategy for subdividing a pixel and counting the number of subpixels covered (at least to a significant degree) by a line, as part of a line-drawing algorithm using unweighted area sampling.

3.33 Create tables with various decreasing functions of the distance between pixel center and line center. Use them in the antialiased line algorithm of Fig. 3.62. Contrast the results produced with those produced by a box-filtered line.

3.34 Generalize the antialiasing techniques for lines to polygons. How might you handle nonpolygonal boundaries of curved primitives and characters?

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

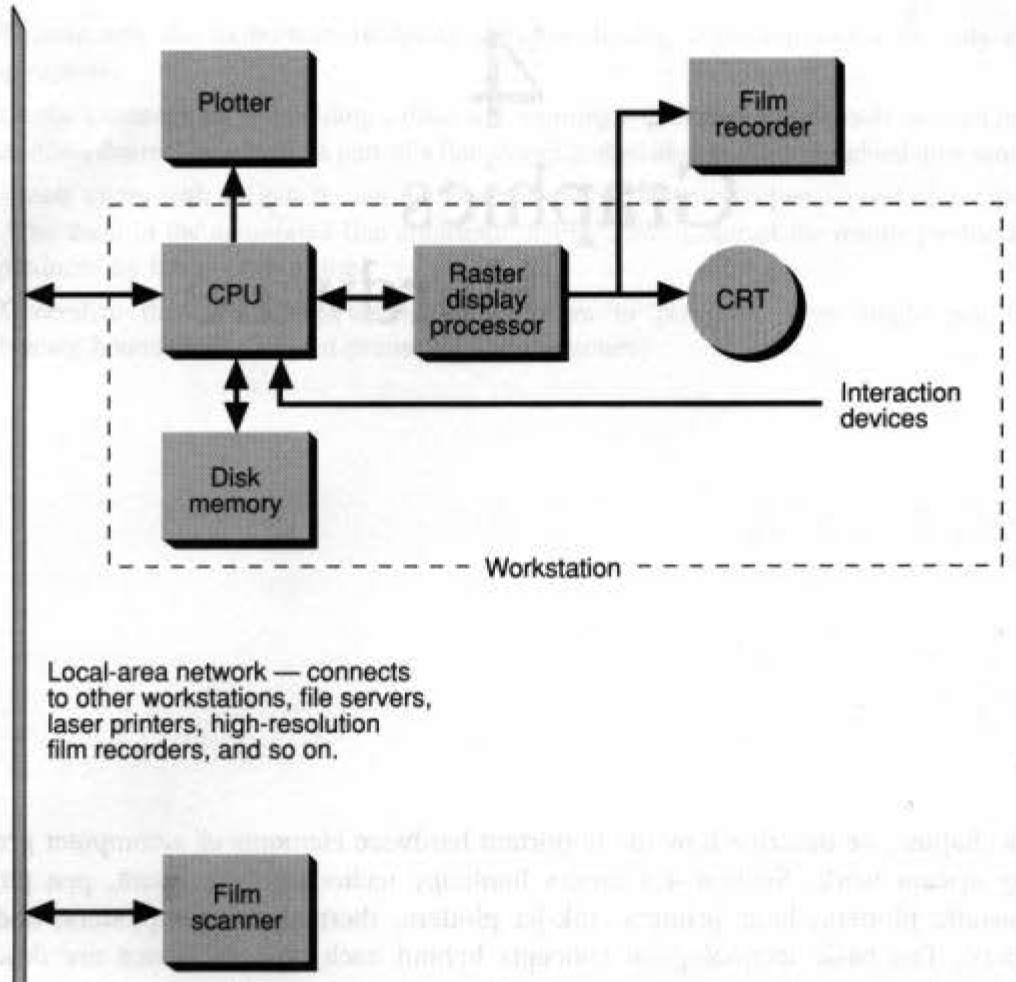

Plotter
Film recorder
CPU
Raster display processor
CRT
Interaction devices
Disk memory
Workstation
Local-area network — connects to other workstations, file servers, laser printers, high-resolution film recorders, and so on.
Film scanner

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 *spót 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.

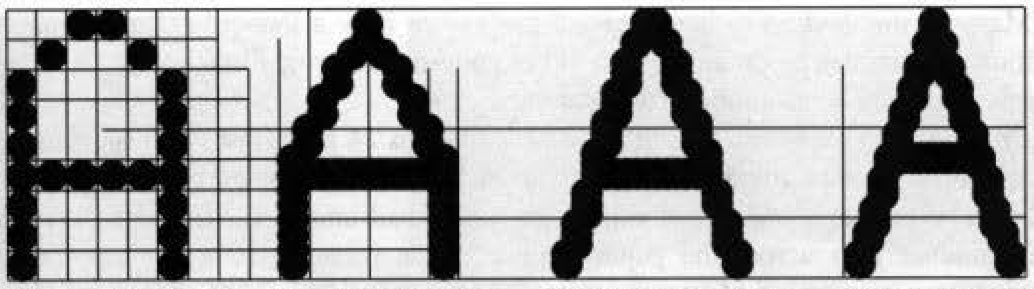

(a) Interdot spacing equal to dot size
(b) Interdot spacing one-half dot size
(c) Interdot spacing one-third dot size
(d) Interdot spacing one-quarter dot size

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.

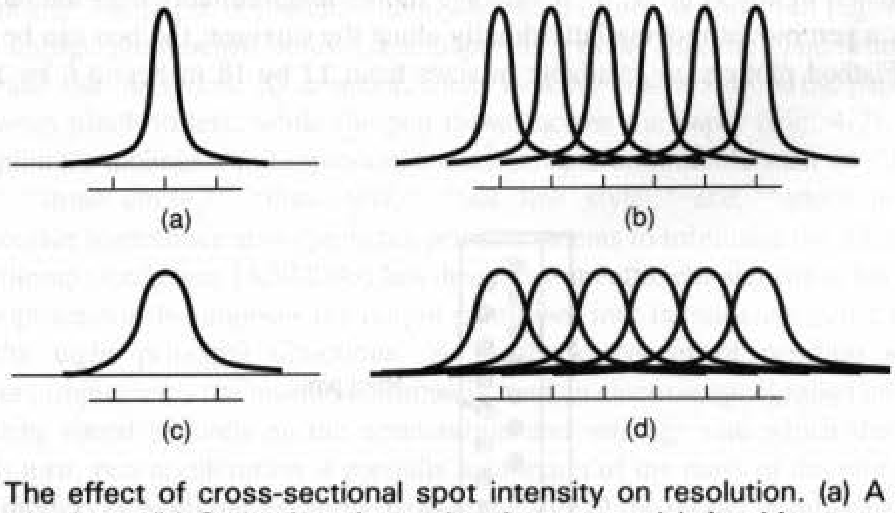


Fig. 4.3 The effect of cross-sectional spot intensity on resolution. (a) A spot with well-defined edges. (b) Several such overlapping spots. (c) A wider spot, with less height, since the energy is spread out over a larger area; its edges are not well defined, as are those in (a). (d) Several of these 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.

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

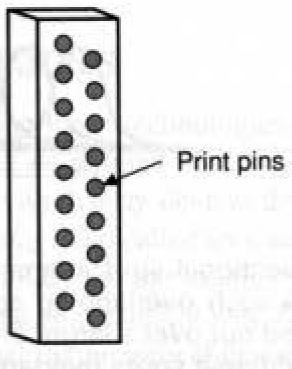


Fig. 4.4 A dot-matrix print head with two columns of print pins, offset vertically by half the interpin spacing to increase resolution.

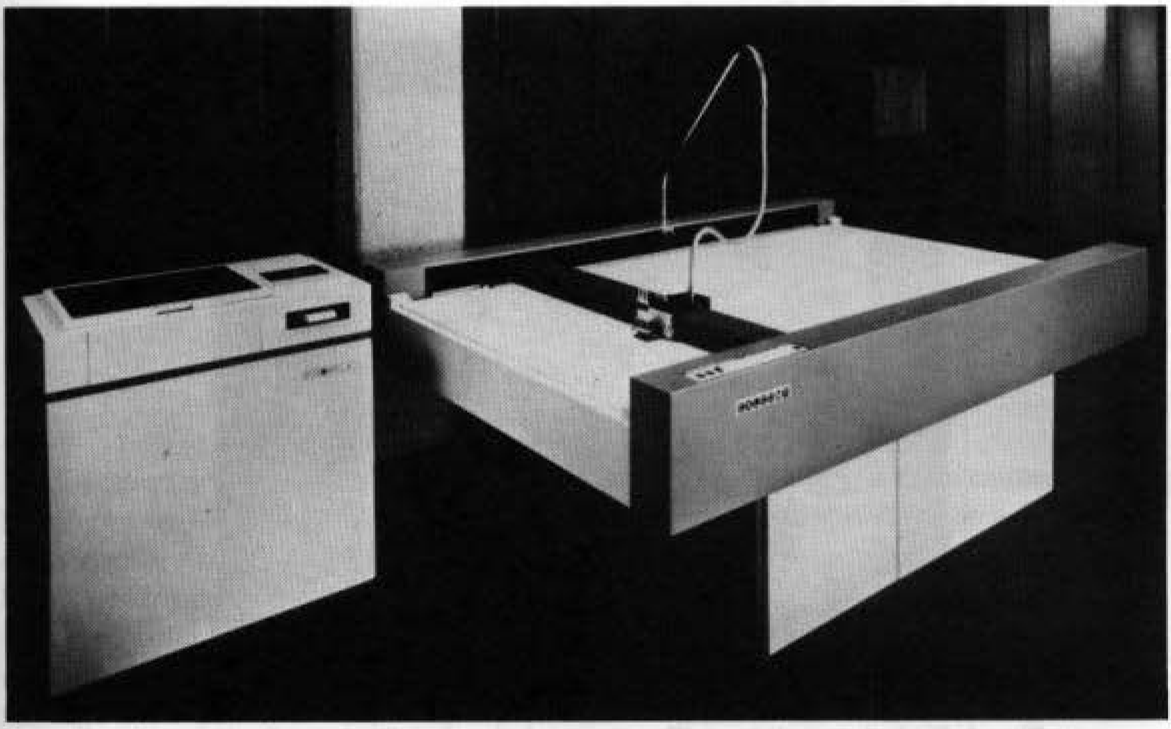


Fig. 4.5 A flatbed plotter. (Courtesy of CalComp–California Computer Products, Inc.)

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

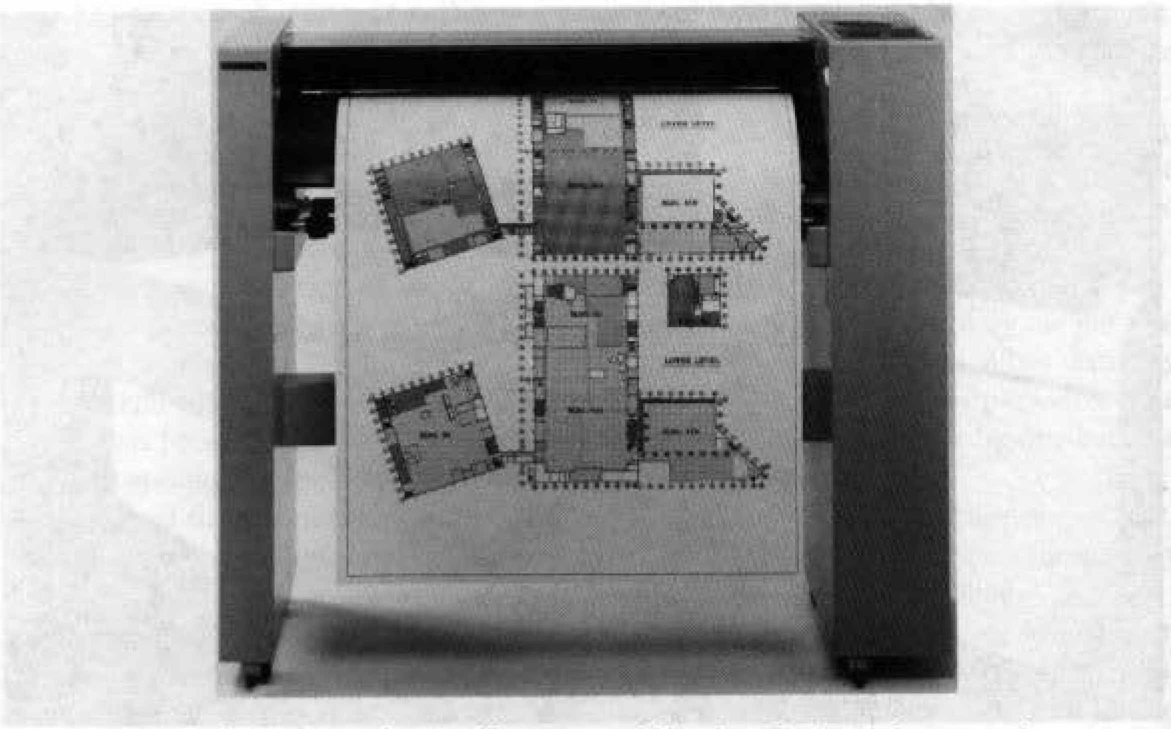


Fig. 4.6 A drum plotter. (Courtesy of Hewlett-Packard Company.)

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.

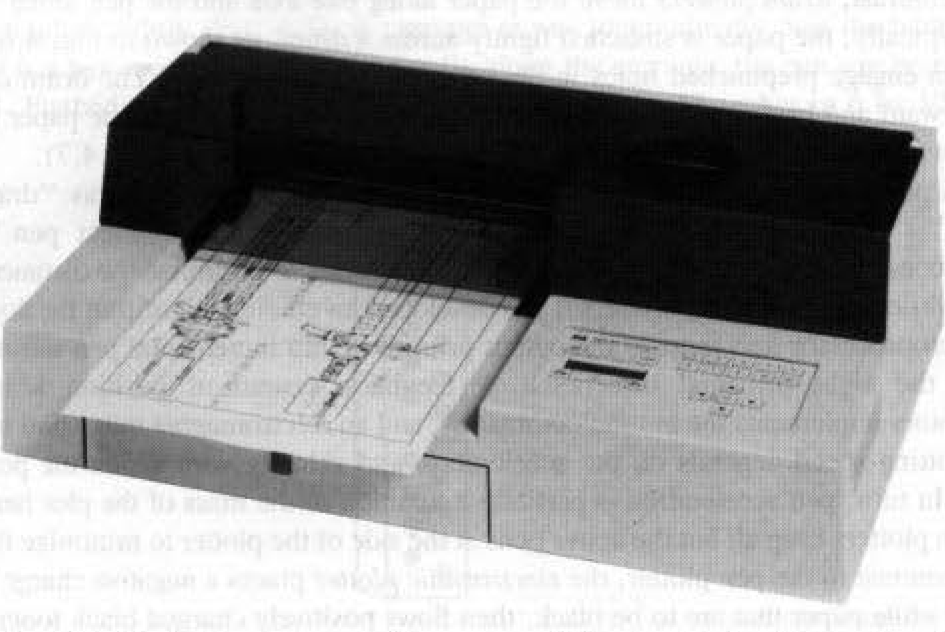


Fig. 4.7 A desk-top plotter. (Courtesy of Hewlett-Packard Company.)

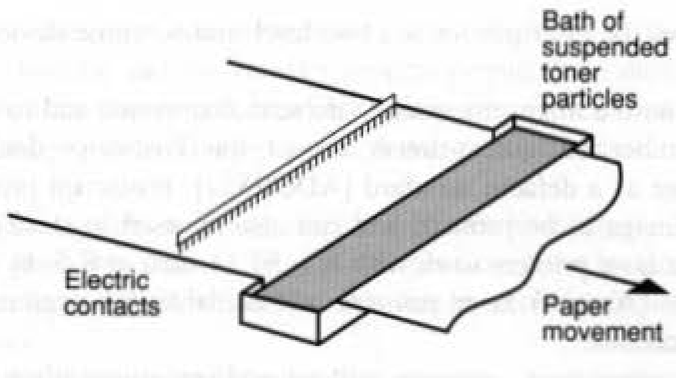


Fig. 4.8 Organization of an electrostatic plotter.

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

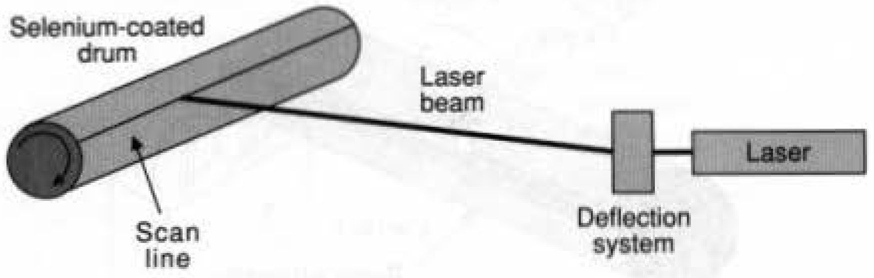


Fig. 4.9 Organization of a laser printer (the toner-application mechanism and the paper feeder are not shown).

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.

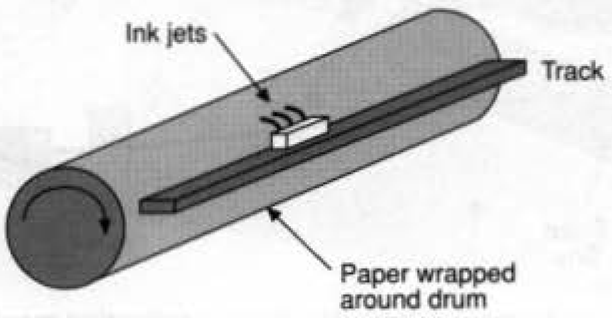


Fig. 4.10 A rotary ink-jet plotter.

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

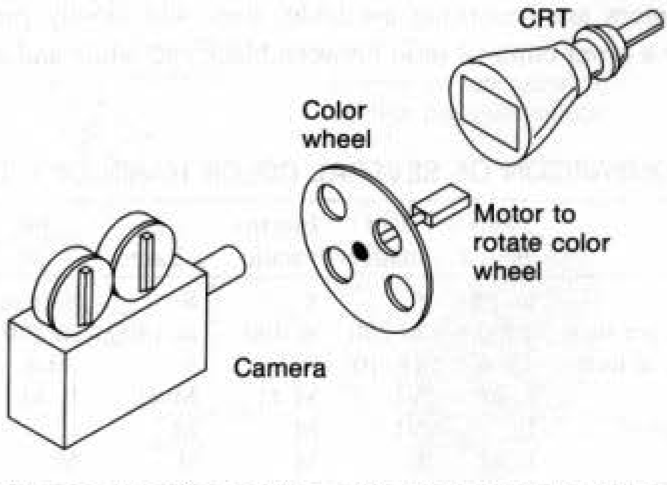


Fig. 4.11 A film recorder for making color photographs using colored filters.

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.