

Fundamentals of Interactive Computer Graphics

J. D. FOLEY • A. VAN DAM



THE SYSTEMS PROGRAMMING SERIES



Fundamentals of
Interactive Computer Graphics

FOLEY
VAN DAM



ADDISON
WESLEY

14468

Look for these other books in the Addison-Wesley Systems Programming Series

An Introduction to Operating Systems by Harvey M. Deitel, 1982

Conceptual Structures: Information Processing in Mind and Machine by John F. Sowa, 1982

An Introduction to Database Systems, Third Edition by C. J. Date, 1981

Database Security and Integrity by Eduardo N. Fernandez, Rita C. Summers, and Christopher Wood, 1981

Operating Systems by Harold Lorin and Harvey M. Deitel, 1981

The Mathematical Foundations of Programming by Frank S. Beckman, 1980

Coded Character Sets by Charles E. Mackenzie, 1979

Structured Programming: Theory and Practice by Richard C. Linger, Harlan D. Mills, and Bernard I. Witt, 1979

Modeling and Analysis: An Introduction to System Performance Evaluation Methodology by Hisashi Kobayashi, 1978

Communications Architecture For Distributed Systems by Rudolph J. Cypser, 1978

The Environment for Systems Programs by Frederic G. Withington, 1978

Compiler Design Theory by Philip M. Lewis, II, Daniel J. Rosenkrantz, and Richard E. Stearns, 1976

Recursive Programming Techniques by William H. Burge, 1975

Sorting and Sort Systems by Harold Lorin, 1975

The Structure and Design of Programming Languages by John E. Nichols, 1975

The Program Development Process: Volume I, The Individual Programmer by Joel D. Aron, 1975

ADDISON-WESLEY PUBLISHING COMPANY

ISBN 0-201-14468-9

THE
SYSTEMS
PROGRAMMING
SERIES

Fundamentals of Interactive Computer Graphics

J. D. FOLEY
A. VAN DAM



Fundamentals of
Interactive Computer Graphics

FOLEY
VAN DAM



ADDISON
WESLEY

14468

ISBN 0-201-14468-9

Fundamentals of Interactive Computer Graphics

JAMES D. FOLEY

The George Washington University

ANDRIES VAN DAM

Brown University



Addison-Wesley Publishing Company

Reading, Massachusetts • Menlo Park, California

London • Amsterdam • Don Mills, Ontario • Sydney

Sponsoring Editor: William B. Gruener
Production Editor: Rima Zolina
Designer: Herb Caswell
Illustration: ANCO/Boston
Cover Design: Richard Hannus
Art coordinator: Dick Morton

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

Library of Congress Cataloging in Publication Data

Foley, James D. 1942-
Fundamentals of interactive computer graphics.

(The Systems programming series)

Bibliography: p.

Includes index.

1. Computer graphics. 2. Interactive computer systems. I. Van Dam, Andries, 1938- joint author.

II. Title.

T385.F63 001.64'43 80-24311

ISBN 0-201-14468-9

Copyright © 1982 by Addison-Wesley Publishing Company, Inc. Philippines copyright 1982 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. Published simultaneously in Canada. Library of Congress Catalog Card No. 80-24311

ISBN 0-201-14468-9
ABCDEFGHIJ-HA-8987654321

THE SYSTEMS PROGRAMMING SERIES



*The Program Development Process
Part I—The Individual Programmer

Joel D. Aron

The Program Development Process
Part II—The Programming Team

Joel D. Aron

*Mathematical Foundations of
Programming

Frank Beckman

*Structured Programming: Theory
and Practice

Richard C. Linger
Harlan D. Mills
Bernard I. Witt

*Coded Character Sets: History and
Development

Charles E. Mackenzie

*The Structure and Design of Program-
ming Languages

John E. Nicholls

*The Environment for Systems Programs

Frederic G. Withington

*Communications Architecture for
Distributed Systems

R. J. Cypser



*An Introduction to Database Systems,
Third Edition

C. J. Date

*Database Security and Integrity

Eduardo B. Fernandez
R. C. Summers
C. Wood

*Fundamentals of Interactive
Computer Graphics

James Foley
Andries Van Dam

*Compiler Design Theory

Philip M. Lewis II
Daniel J. Rosenkrantz
Richard E. Stearns

*Sorting and Sort Systems

Harold Lorin

*Operating Systems

Harold Lorin
Harvey M. Deitel



*Recursive Programming Techniques

William Burge

*Modeling and Analysis: An Introduc-
tion to System Performance Evalua-
tion Methodology

Hisashi Kobayashi

Conceptual Structures: Information
Processing in Mind and Machines

John F. Sowa

*Published

IBM EDITORIAL BOARD

Joel D. Aron
Richard P. Case, Chairman
Gerhard Chroust
Robert H. Glaser
Charles L. Gold
Paul S. Herwitz

James P. Morrissey
George Radin
David Sayre
Heinz Zemanek
William B. Gruener (Addison-Wesley)

The field of s
programmers
utilitarian sys
Programming
solutions to pr
associates. In
necessary to
accessible to
that enough u
his recommen
as a long ter
techniques th

The Serie
contents of ea
and do not
organized for
is organized
volumes, mo
ized theory in
of the novice

Taken to
programming
discipline.

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.

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. Further, the Series is organized in three levels: broad introductory material in the foundation volumes, more specialized material in the software volumes, and very specialized theory in the computer science volumes. As such, the Series meets the needs of the novice, the experienced programmer, and the computer scientist.

Taken together, the Series is a record of the state of the art in systems programming that can form the technological base for the systems programming discipline.

The Editorial Board

Preface

Interactive computer 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 which 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. Pictorial communication is a medium that is both natural and efficient to human beings and yet is sufficiently precise for computer manipulation. Most people enjoy interacting graphically more than they do the more traditional and more limited alphanumeric communication techniques. Interactive graphics can be used to understand complex phenomena, to design technological artifacts, and to amuse—it is an extremely versatile, aesthetically pleasing and instructive medium.

The purpose of this book is to provide both a tutorial and a reference source for readers interested in the many aspects of modern interactive graphics: hardware, software, data structures, mathematical manipulation of graphical objects, the user interface, and the fundamental implementation algorithms. No prior background in graphics is needed, although a basic background in programming and data structures and some familiarity with computer architecture are assumed. The book is designed so that the reader can progress through a carefully designed sequence of presentation shells, starting with simple, generally applicable fundamentals and ending with the more complex and more specialized subjects.

Shell One: Basics.

In the field of graphics, as in most others in computer science, one learns best by doing. Therefore, after a general introduction in Chapter 1, we start immediately with graphics application programming in Chapter 2. This will prepare students at the earliest possible moment to write their own programs and thus deepen their understanding of the material and the issues presented. Since interaction is such a vital part of interactive graphics, we stress good program structures for user-friendly interactive dialogue design, and show, through a sequence of progressive examples, how picture plotting and input handling can be naturally integrated to achieve such a design. For our programming environment, we use Pascal and a subset of the "Core" Graphics subroutine package proposed as a standard by ACM SIGGRAPH [GSPC 79]. Chapters 3 and 4 treat fundamental hardware and software concepts to show how the simple graphics package of Chapter 2 can be easily implemented. Windowing, clipping, segmentation, and logical interaction handling are described. Chapter 5 treats specific techniques for using input devices, while Chapter 6 discusses design principles for the user-computer interface.

Shell Two: Mathematics, data structures, and display architecture.

We deepen our exploration of fundamentals by starting with the basic mathematics of two-dimensional and three-dimensional geometric and viewing transformations in Chapters 7 and 8. Chapter 9 provides a more detailed treatment of the geometric transformations and data structures useful for modeling objects to be displayed, especially those which can be represented in a hierarchical form. Chapter 10 shows how display processor architecture is evolving, especially with respect to functional distribution.

Shell Three: Modern raster technology.

Chapter 11 extends the basic raster techniques discussed in Chapter 2 to cover relevant device-independent algorithms in use today and gives a picture of raster graphics software. Chapter 12 provides an overview of current raster hardware technology.

Shell Four: Making realistic synthetic photographs.

This is the most advanced shell and covers the part of computer graphics that is both the newest and, to many people, the most exciting. Chapter 13 treats fundamental techniques for representing three-dimensional surfaces, and Chapter 14 introduces general strategies for making realistic pictures of three-dimensional objects. The classic problem of removing surfaces of solid objects hidden by other surfaces is addressed in Chapter 15. Chapter 16 discusses shading and texturing algorithms and various lighting models, and Chapter 17 treats models for representing and specifying color.

The layered structure of the book will, we hope, make it useful in a variety of settings. An introductory one-semester undergraduate course can use much of the material in the first three shells, with a sprinkling of introductory material from the fourth. A graduate course can progress more rapidly through the first three shells and concentrate more on the advanced material of shell four. Application programmers or students who have had a first course in graphics and are familiar with classi-

cal vector gra
before studyi
tic-looking sh

We are g
after we spent
outdated by r
nal ACM SIG
Bergeron, Ing
many ideas in
able draft "C
us the benefit
major impact
Bergeron, Jac
Feiner, Alan
Metrick, Nori
bert, Alvy Ra
Gerry Weil, T
one or more o
checked out th
ter 2 to check
script, while t
Doell, Sandy
typed by Jacqu
neros. We th
Zolina, for th
organizations
Lee Metrick, I
book. As is cu
the early draft
ance. Finally,
pressure on us
with our man
Orchid Room
ing marathon

Washington, I
Providence, R

August 1981

cal vector graphics can use the early shells to review basic notions and techniques before studying the newer material on raster graphics and the production of realistic-looking shaded and colored pictures.

We are grateful to the many people who have helped make this book possible, after we spent too many years seeing our less than full-time writing effort constantly outdated by rapidly moving developments in the field. Our colleagues on the original ACM SIGGRAPH Graphics Standards Planning Committee, Peter Bono, Dan Bergeron, Ingrid Carlbom, Jim Michener, and Elaine Sonderegger, helped us put many ideas into perspective as we pursued our common goal of producing a workable draft "Core" standard. We are especially indebted to Jim Michener for giving us the benefit of a superb, painstakingly detailed, red-pencil review which had a major impact on the content, style, and organization of this book. We thank Dan Bergeron, Jack Bresenham, Dick Bulterman, Ingrid Carlbom, John Dill, Steve Feiner, Alan Frieden, Bob Heilman, Janet Incerpi, Abid Kamran, Jeff Lane, Lee Metrick, Norm Meyrowitz, Aragam Nagesh, Joe Pato, Jean Schweitzer, John Sibbert, Alvy Ray Smith, Jagan Sud, Jim Templeman, Barry Trent, Victor Wallace, Gerry Weil, Turner Whitted, and Patricia Wenner, all of whom critically reviewed one or more of the chapters. Barry Trent implemented SGP and he and Gerry Weil checked out the Pascal programs; Adam Seidman implemented *LAYOUT* in chapter 2 to check it out. Katrina Avery did her usual fine editorial work on the manuscript, while typing and editing were skillfully done by herself, Mary Agren, Karen Doell, Sandy Ballentine, Marilyn Henry, and Virginia Edwards. Earlier drafts were typed by Jacqueline Bowman, Betty Kirschbaum, Hilda Wagstaff, and Patricia Cisneros. We thank the Addison-Wesley staff, especially Dick Morton and Rima Zolina, for their patience and professional help, and the many individuals and organizations who provided us with photographs and illustrations. In particular, Lee Metrick, Dave Shuey, and Rick Thorne prepared illustrations especially for the book. As is customary in such endeavors, students in our graphics courses suffered the early drafts of our chapters, and we are grateful for their comment and perseverance. Finally, we are grateful to our former students and colleagues for keeping pressure on us, and especially to our families for keeping the faith and putting up with our many prolonged absences and excuses. (And a final thanks to the Thai Orchid Room for its "crispy spicy fish" which nourished us during our many writing marathons!)

Washington, D.C.
Providence, R.I.

J.D.F.
A.v.D.

August 1981

Contents

CHAPTER 1 **WHAT IS INTERACTIVE GRAPHICS?**

1.1	Introduction	1
1.2	Image Processing as Picture Analysis	3
1.3	Advantages of Interactive Graphics	5
1.4	Some Representative Uses of Computer Graphics	8
1.5	Classification of Applications	11
1.6	Interactive Graphics in the Future: The Normal Mode of Interaction	14
1.7	Brief History of Computer Graphics	18
1.8	Overview of the Programmer's Model of Interactive Graphics	24
	Exercises	28

CHAPTER 2 **BASIC INTERACTIVE GRAPHICS PROGRAMMING**

2.1	Models, Picture Description, and Interaction	29
2.2	Introduction to the Simple Graphics Package	35
2.3	Graph Plotting	36
2.4	Windows and Clipping	40
2.5	Segmentation	43
2.6	Simple Graph	44
2.7	Viewports	46
2.8	Character Strings	49
2.9	Summary of SGP's Picture-Making Facilities	50
2.10	Interactive Graphics Program—Layout of Symbols	50
2.11	Generating a Picture from the Data Structure	51
2.12	Interactive Programming	55

xvi Contents

2.13	Mainline Using Buttons for Function Invocation	59
2.14	Using a Menu for Function Invocation	61
2.15	Defining a Title	68
2.16	Adding Symbols to the Data Structure	69
2.17	Deleting Symbols	77
2.18	Changing View by Panning and Zooming	78
2.19	Adding Solid Areas for Raster Displays	87
2.20	Alternative Implementation	88
2.21	Summary	89
	Exercises	90

CHAPTER 3 GRAPHICS HARDWARE

3.1	Output-Only Technology	95
3.2	Display Technologies	102
3.3	Random-Scan Display Processing Unit	112
3.4	Input Devices for Operator Interaction	127
3.5	Raster-Scan Display Processing Unit	129
	Exercises	135

CHAPTER 4 IMPLEMENTATION OF A SIMPLE GRAPHICS PACKAGE (SGP)

4.1	Overview of SGP	137
4.2	Viewing Operation	143
4.3	DPU Program Code Generation	155
4.4	CPU-DPU Synchronization	168
4.5	Interaction Handling	169
4.6	Segmented DPU Programs for Other Devices	171
4.7	Error Handling	173
4.8	Device-Independent Graphics	174
4.9	Summary	177
	Exercises	179

CHAPTER 5 INTERACTION DEVICES AND TECHNIQUES

5.1	Logical Classes of Devices and Techniques	183
5.2	Physical Interaction Devices	184
5.3	Techniques for Simulating Logical Devices	197
5.4	Interaction Techniques	208
	Exercises	214

CHAPTER 6 THE DESIGN OF USER-COMPUTER GRAPHIC CONVERSATIONS

6.1	The Language Analogy	218
6.2	The Language Model	220

6.3	Design
6.4	The Design Process
	Exercises

CHAPTER 7 GEOMETRIC TRANSFORMATIONS

7.1	2D Transformations
7.2	Homogeneous Coordinates
	2D Transformations
7.3	Compositions
7.4	Efficient Algorithms
7.5	Matrix Representations
7.6	Compositions
7.7	Translations
	Exercises

CHAPTER 8 VIEWING IN 3D

8.1	Projection
8.2	The Viewing Process
8.3	Specifications
8.4	Calculations
8.5	Clipping
8.6	Image Generation
8.7	3D Viewing
8.8	Summary
	Exercises

CHAPTER 9 MODELING AND RENDERING

9.1	What is Modeling?
9.2	Geometric Modeling
9.3	Object Modeling
9.4	Transformations
9.5	Generalization
9.6	How to Model
9.7	Multiresolution
9.8	Passing Data
9.9	Pre- and Post-Processing
9.10	Combining Models
9.11	Clipping
9.12	Extents
9.13	Object Visibility
9.14	Summary
	Exercises

6.3	Design Principles	222
6.4	The Design Process	239
	Exercises	243

CHAPTER 7 GEOMETRICAL TRANSFORMATIONS

7.1	2D Transformations	245
7.2	Homogeneous Coordinates and Matrix Representation of 2D Transformations	249
7.3	Composition of 2D Transformations	252
7.4	Efficiency Considerations	254
7.5	Matrix Representation of 3D Transformations	255
7.6	Composition of 3D Transformations	258
7.7	Transformation as a Change of Coordinate Systems	262
	Exercises	265

CHAPTER 8 VIEWING IN THREE DIMENSIONS

8.1	Projections	268
8.2	The Mathematics of Planar Geometric Projections	274
8.3	Specifying an Arbitrary 3D View	279
8.4	Calculating the Planar Geometric Projections	284
8.5	Clipping Against a Canonical View Volume	295
8.6	Image Transformations	297
8.7	3D Viewing in Simple Graphics Package (SGP)	302
8.8	Summary	316
	Exercises	317

CHAPTER 9 MODELING AND OBJECT HIERARCHY

9.1	What is a Model?	319
9.2	Geometric Models and Object Hierarchy	323
9.3	Object Placement and Instance Transformations	325
9.4	Transforming Objects with the Current Transformation Matrix	328
9.5	Generalizing to Two-Level and n -Level Hierarchies	334
9.6	How Can Hierarchy Be Encoded?	346
9.7	Multilevel Correlation	362
9.8	Passing Attributes	365
9.9	Pre- and Postmultiplication	366
9.10	Combining Model Traversal with the Viewing Operation	368
9.11	Clipping in Master Coordinates	369
9.12	Extents	375
9.13	Object Windows and Instance Rectangles	381
9.14	Summary	386
	Exercises	387

CHAPTER 10

ADVANCED DISPLAY ARCHITECTURE

10.1	Introduction	391
10.2	Background	391
10.3	The Simple Refresh Display	394
10.4	Vector Transformations and Clipping in Image Space	400
10.5	High-Performance Displays with Modeling Transformations and Viewing Operation	403
10.6	Functional Model for High-Performance Architectures	406
10.7	The Evans & Sutherland PS300—Another Turn on the Wheel	418
10.8	Extensions for Raster Graphics—Megatek 7200	421
10.9	Multiprocessor and Host-Satellite Graphics	424
	Summary	428
	Exercises	428

CHAPTER 11

RASTER ALGORITHMS AND SOFTWARE

11.1	Introduction	431
11.2	Scan-Converting Lines	432
11.3	Scan-Converting Characters	439
11.4	Scan-Converting Circles	441
11.5	Region Filling	446
11.6	Polygon Clipping	450
11.7	Scan-Converting Polygons	456
11.8	Unsegmented Graphics Subroutine Package with Temporal Priority	463
11.9	Segmented Graphics Subroutine Package with Priority	470
	Exercises	476

CHAPTER 12

DISPLAY ARCHITECTURE

12.1	Introduction	479
12.2	A Simple Raster Display System	479
12.3	Programming the Raster Display	486
12.4	The Image Display System	488
12.5	Alternative Raster Display Architectures	495
12.6	Raster Architecture for Personal Computers	501
	Exercises	503

CHAPTER 13

REPRESENTATION OF 3D SHAPES

13.1	Introduction	505
13.2	Polygon Meshes	507
13.3	Consistency of Polygon Mesh Representations	510
13.4	Plane Equations	512
13.5	Parametric Cubic Curves	514

13.6	Pa
13.7	Tr
13.8	Ca
	Ex

CHAPTER THE QUI

14.1	In
14.2	Te
14.3	St
14.4	Ap

CHAPTER ALGORIT

15.1	Im
15.2	Sir
15.3	Av
15.4	Th
15.5	Th
15.6	Sc
15.7	Ar
15.8	Al
15.9	Al
	Ex

CHAPTER SHADING

16.1	Int
16.2	Di
16.3	Sp
16.4	Po
16.5	Sh
16.6	Lig
16.7	Su
	Ex

CHAPTER INTENSIT

17.1	Int
17.2	Ac
17.3	Ch
17.4	Co
17.5	Re

13.6	Parametric Cubic Surfaces	523
13.7	Transforming Curves and Patches	530
13.8	Calculation of Points on a Bicubic Surface	531
	Exercises	536

CHAPTER 14

THE QUEST FOR VISUAL REALISM

14.1	Introduction	539
14.2	Techniques for Displaying Depth Relationships	542
14.3	Stereopsis	546
14.4	Approaches to Visual Realism	550

CHAPTER 15

ALGORITHMS FOR REMOVING HIDDEN EDGES AND SURFACES

15.1	Introduction	553
15.2	Simplifying Depth Comparisons: The Perspective Transformation	554
15.3	Avoiding Depth Comparisons: Extents	556
15.4	The Depth-Sort Algorithm	558
15.5	The z -Buffer Algorithm	560
15.6	Scan-Line Algorithms	561
15.7	Area-Subdivision Algorithms	565
15.8	Algorithm Efficiency	569
15.9	Algorithms for Curved Surfaces	570
	Exercises	572

CHAPTER 16

SHADING MODELS

16.1	Introduction	575
16.2	Diffuse Reflection and Ambient Light	575
16.3	Specular Reflection	577
16.4	Polygon Mesh Shading	580
16.5	Shadows	584
16.6	Light-Transmitting Surfaces	585
16.7	Surface Detail	588
	Exercises	590

CHAPTER 17

INTENSITY AND COLOR

17.1	Introduction	593
17.2	Achromatic Color: Intensity	593
17.3	Chromatic Color	602
17.4	Color Models for Raster Graphics	611
17.5	Reproducing Color Hard Copy	620

xx Contents

17.6 Color Harmony	621
17.7 Using Color in Interactive Graphics	621
Exercises	622
BIBLIOGRAPHY	625
INDEX	655

1.1 INTRODUCTION

We introduce the concepts of interactive graphics that are familiar to most people.

- High-speed graphics: High-speed graphics are used by people (Fig. 1.1).
- Drum or disk graphics: Two-dimensional and three-dimensional graphics, such as flowcharts, are used.
- Film or video graphics: Film or video graphics are used to show pictures of real objects, such as graphs (Color Plate 1). They are used from science fiction to sports.
- Storage tube graphics: Storage tube graphics are used to show pictures of real objects, such as thumbwheels, and are used for plotting of text. The screen is erased and a new line is drawn at a time. It takes several minutes to generate a picture, but it is as precise as if it were drawn by hand.
- Hobby (hobby) graphics: Hobby graphics are used for play, used for showing pictures typical of a game, such as letters, lines, and numbers, and are used for communicating clear information.

ming are bound to change, perhaps fundamentally, as we move into the age of interactive graphics on powerful personal computers. Nonetheless we feel that Core terminology and Core implementations are rapidly becoming *de facto* (though interim) standards. We will therefore introduce graphics application programming in the next chapter by using a subset of the Core.

1.8 OVERVIEW OF THE PROGRAMMER'S MODEL OF INTERACTIVE GRAPHICS

The conceptual framework on which the Core is based is shown schematically in Fig. 1.13. We will describe it briefly here and return to it in later chapters for a more detailed treatment. The hardware component is a host computer that drives a *display device*, also called the *display unit*, *display*, or *graphics terminal* (such as in Figs. 1.10 through 1.12). The display itself consists of an *output* component (the *display screen*, or *view surface*, on which pictures are displayed) and an *input*, or *interaction component* (Fig. 1.14). This input component typically consists of a set of *logical devices*, including *keyboard devices* such as an alphanumeric keyboard for entering text, a *function button device* for invoking predefined options or functions, a *picking device* such as a light pen or data tablet stylus for indicating picture components on the screen, *valuators* such as control dials and levers for entering scalar values, and an *xy position indicator* or *locator*, such as a cross-hair thumbwheel, lever, or joystick (see Chapters 3 through 5 for details on all these devices). Note that one physical device may be used to implement one or more of these logical input functions.

The software consists of three components. The first is the *application program*; it stores into and retrieves from the second component, the *application data structure/data base*, and sends graphics commands to the third component—the *graphics system*.

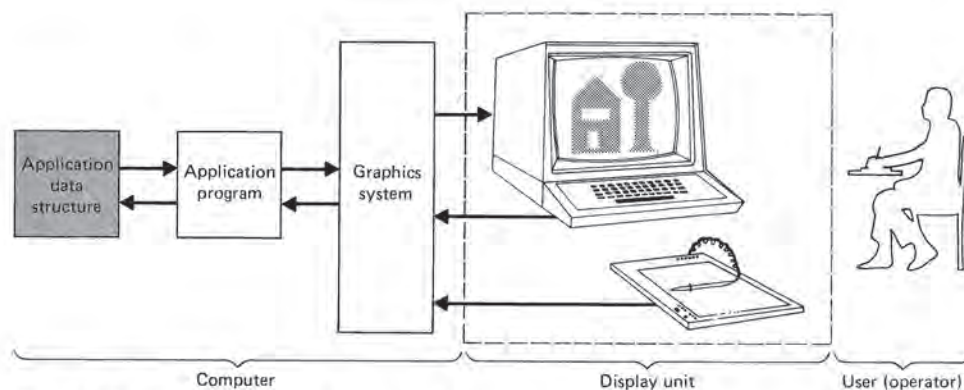


Fig. 1.13 The programmer's model of interactive graphics.

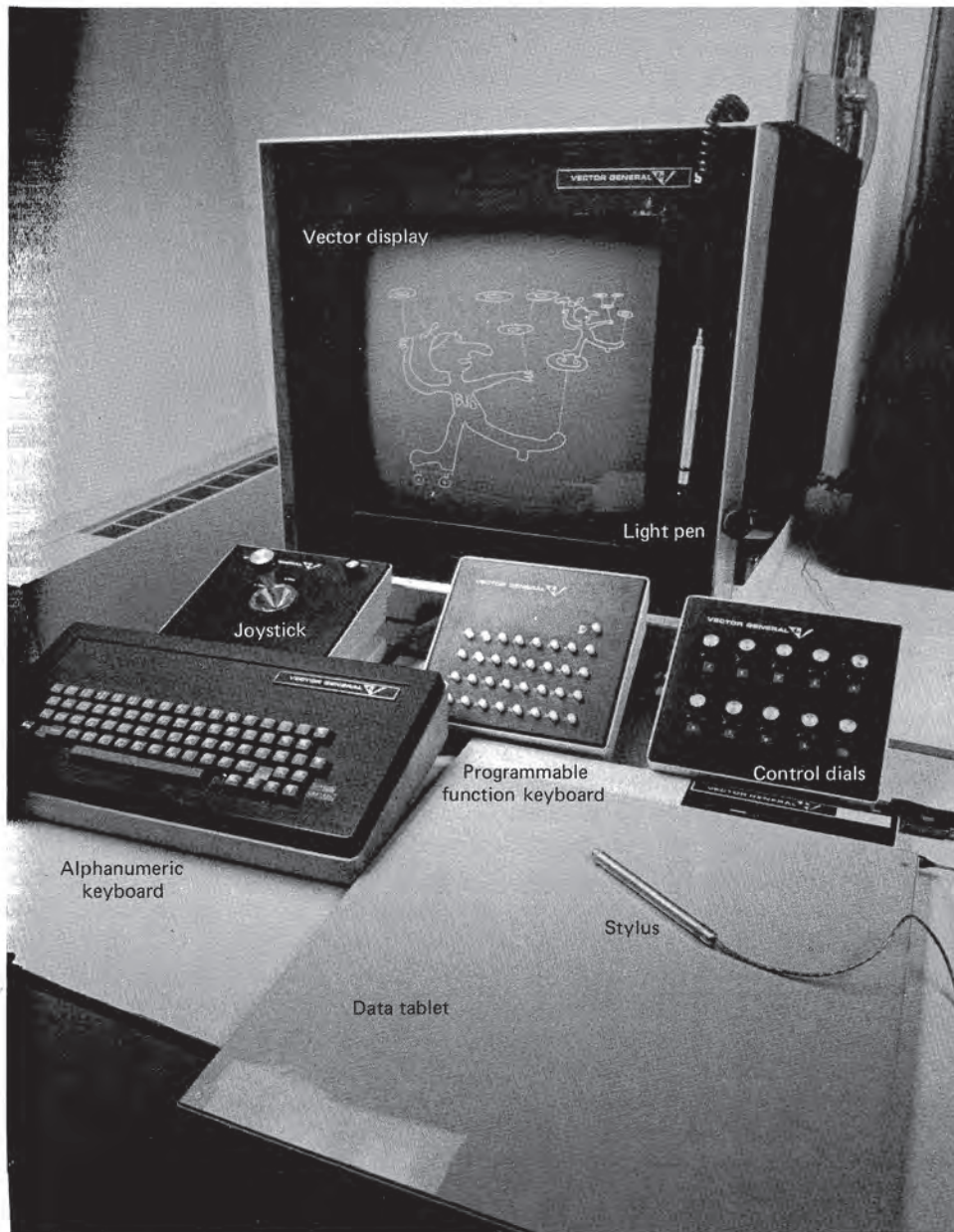


Fig. 1.14 Display console with common input devices.

The data structure holds descriptions of real or abstract objects whose pictures are to appear on the screen. The data structure thus stores all of the pertinent information for such "objects" as circuits, flowcharts, buildings, mathematical or statistical functions, airplane fuselages, molecules, nuclear-reactor models, and 3D landscapes and vehicles used for flight simulators.

The object's description typically contains *geometric* coordinate data that define the shape of components of the object, object *attributes* such as line style, color, or even surface texture, and *connectivity* relationships and positioning data that define how the components fit together. Often there is also nongeometric textual or numeric "property" information useful to a post-processing program and/or the interactive user. Examples of such data for computer-aided design applications include price and supplier data, thermal, mechanical, or electrical properties, and mechanical or electrical tolerances. Note that general-purpose database management systems are increasingly being used for the data structure function.

The application program describes the 2D or 3D geometry of the object whose picture is to be viewed on the view surface to the *graphics system*, which typically contains a collection of *output* plotting subroutines compatible with a high-level language such as FORTRAN or Pascal. This subroutine package drives the specific output device(s) and causes the device to display the picture, usually from the picture's display list representation in the refresh buffer that the package has just created.

The application program uses the graphics system much as it uses the input-output subsystem of the operating system to read and write records in files. The input-output subsystem maintains a file directory, organizes records on disks and tapes, and shields the application programmer from having to know the many device-dependent parameters and conditions. Similarly, the graphics system shields the application programmer from needing to know the specific low-level architecture of the display processor and the *xy* coordinate system of the physical screen.

While picture plotting is handled by the graphic system's output routines, input handling is controlled by its *input* routines that pass user-supplied input data to the application program as part of an interaction sequence. Thus the application program can also ask the graphic system to read values from (i.e., to *sample*) the control dials or locator input devices. Alternatively, the application program can put itself to sleep after having asked an input routine to wake it up when an *interrupt-generating* input device, such as a keyboard or stylus pick, is activated by the user.

With these input values, the application program can change its operating mode or state. For example, it can enter the procedure that corresponds to one of a number of possible commands listed in a "menu" on the screen, in response to the user's menu selection. In addition to executing such a change in flow of control, the program can simply alter a data value which then modifies the appearance of an object or changes an associated property.

The *output transformation* from object to picture and the *input transformation* from user action to object are treated in considerably more detail in Chapter 2 (from the application programmer's point of view) and in Chapter 4 (from the package

implementer's point of view). With this preliminary understanding, however, we can now show how graphics application programs differ from "conventional" programs.

First of all, the two- and three-dimensional geometry of graphical objects represents a different type of data, one with its own rapidly evolving body of mathematics and algorithms. This is especially true in the exciting area of making realistic-looking pictures of three-dimensional objects (see Chapters 14–17). Second, interactive graphics programs are different from batch programs in that they tend to be *event-driven*; that is, they wait for the user to do something, react appropriately, and wait again. Essentially, the dialogue consists of a simple loop:

```
repeat
  provide a choice/pose a question;
  wait for the user to respond with one of  $n$  allowed values;
  case—branch on the answer to the appropriate procedure for that answer;
until stop {user responds with STOP}
```

Third, the use of a high-level device-independent graphics package makes interactive plotting and manipulation of simple graphs, charts, and drawings very straightforward, as will be shown in Chapter 2. For more complex applications, however, good graphics programming practice requires some understanding of a number of areas not always properly mastered by application programmers:

1. Construction of dialogue and interaction "languages," with proper attention to expressive power, consistency, good "human factors" of the human interface for ease and speed of interaction, and graceful and informative error handling (Chapter 6);
2. For three-dimensional graphics, the proper use of the mathematics of geometric transformation (Chapter 7) and of shape and surface description (Chapter 13);
3. For realistic pictures, hidden surface removal and the use of color, lighting, and shading models (Chapters 14–17);
4. Manipulation and display of a hierarchy of objects composed of subobjects (Chapter 9);
5. Good software engineering principles for creating a robust, extensible, and maintainable graphics program.

In general it is difficult to graft an interactive graphics interface onto an existing batch program: the style and structure of an interactive graphics application program are usually quite different from those of a batch program, and good program design therefore requires a before-the-fact, top-down analysis of all the required interaction sequences to lead to a properly structured design and implementation. In the next chapter, therefore, we skip the implementation details of graphics hardware and support software and start with graphics programming in order to teach both techniques and good (interactive) programming style by example.

EXERCISES

- 1.1 Make a list of applications of graphics in your organization and try to classify them in terms of the categories of Section 1.5. Can you group the applications into *equivalence classes* of applications that are essentially similar in structure?
- 1.2 Make a high-level user model of a typical interactive graphics application program with which you are somewhat familiar. Identify the major subsystems/modules. How vital is the graphics part; that is, could the application run almost as well if alphanumeric interaction were used? Conversely, how vital is the interactive part—would batch be almost as good?
- 1.3 Take a noninteractive graphics application that you are familiar with and discuss how one might improve its utility, ease of use or cost of use by making it an interactive graphics application. What would be easy to change, what would be hard, and why?
- 1.4 Describe applications that don't need or wouldn't be better with graphical interaction, and state why not. Try to characterize what it is about an application that makes it a good candidate for using interactive graphics.
- 1.5 Take your favorite application area, and try to determine to what extent interactive graphics has been used successfully in it. Use sources such as recent literature in the field, SIGGRAPH conference proceedings, and other graphics publications, as well as talks with experts in the field.
- 1.6 Assume that a picture containing a sequence of 500 connected 10-inch vectors is to be displayed on a refresh display. How many bytes of storage are required to store this picture for a 10×10 -inch vector display that draws continuous lines between two endpoints on a 1024×1024 square grid? Assume a 12-bit "vector draw" opcode and 10 bits for each of the x and y data coordinates, i.e., 4 bytes for the opcode and (x, y) pair per line. Approximately how many bytes are required to store the same picture in raster bit map with the same display and resolution? Which picture would look better and why? Which could be modified faster?
- 1.7 What is a reasonable resolution for a black-and-white bit map raster display refreshed from the memory of a home microcomputer with 64 kbytes of RAM?

effective technique for assisting novices for whom a given prompt may be too terse is to include a **HELP** function that generates detailed instructions for the responses and actions available at a given time.

While prompting tells the user what the options are, feedback occurs after a particular option has been selected and before the next prompt is issued. Cogent feedback is especially important in an interactive graphics environment, and feedback is therefore used after *every* user action in the sample program to demonstrate how the system has interpreted the user's actions (rule 4). As in any graphics system, two different kinds of feedback are given. First, SGP produces for each interaction device an application-independent *echo* which is the immediate response to the device's use; for example, typing a character on an alphanumeric keyboard results in the display of that character, so the user can verify that it is correct. SGP provides default echoing facilities for all input devices. The sample program below relies completely on the default echo facilities, and thus appropriate echoing occurs even though there is no explicit code for it in the program. The second type of feedback, which is higher-level than echoing and application-dependent, shows the application program's response to the user's request. For example, if the user requests that an item be deleted from the model represented in the data structure, it is appropriate that the image of that item also be removed from the screen. This feedback is the responsibility of the application program, which uses SGP to create the desired effect on the screen.

In order to minimize the effect of a user's mistake (rule 5), the program has a **CANCEL** option which allows the user to terminate a partially completed function and, in cases where it makes sense, to "undo" the most recent fully completed function. For consistency, the user completes all functions in the same manner: by use of a **DONE** option. The implementations of the **HELP**, **CANCEL** and **DONE** options are shown later.

Before presenting the details of the interactive aspects of the sample program, we describe the basic interaction facilities of SGP which it uses.

2.12.2 Logical Interaction Devices

A major goal of the input facilities of SGP, as of output devices, is device-independence. Consequently, SGP defines *logical input devices* [WALL76] for communicating between the application program and its user. Program requests for input functions specify a logical device name which SGP maps to the available physical device with the most naturally corresponding characteristics. This mapping of logical to physical devices is analogous to an operating system's mapping of logical unit numbers or logical file names to appropriate physical file storage devices. Physical input devices commonly available include light pens, data tablets, cursor thumbwheels, control dials (potentiometers), joysticks, and alphanumeric keyboards. They are discussed more fully in Chapters 3 and 5. The logical input devices mentioned in Chapter 1 that correspond to one or more of these physical devices include:

- Button, a choice-indicating device;
- Pick, a device for pointing at information displayed on the screen;

- Keyboard, a device for accepting typed alphanumeric character strings;
- Valuator, a device for generating floating-point values;
- Locator, a device for specifying screen coordinates.

2.12.3 Event-Driven Processing

Any graphics programming system must provide the application program with facilities for *synchronizing* user actions with program responses. Interaction between program and user is best understood as an *event-driven* process in which the events correspond to actions performed by the user via input devices.* The program responds appropriately to an event and then waits for the next event to occur. A session with an interactive program (the *user-computer dialogue*) thus consists of user actions alternating with (typically brief) bursts of computation which arise from the user's request or action. Usually the dialogue is dominated by user "think time" during which the program is in the wait state.

The segment of pseudocode below is a schema for a typical interactive application program. Pseudocode is used here and later as an alternative to our Pascal dialect to avoid distracting details; it is shown as free-standing lower case while comments are still in lower case but enclosed in braces. Note that we check to see if the user's actions are allowed, i.e., useful, at this point in the program.

```

initialize, including generating initial picture;
repeat
  WAIT for user-generated event;
  if event_device = case 1 or case 2 or . . . then  {"legal" action}
    case event_device of
      case 1: action 1;
      case 2: action 2;
      .
      .
    end      {case}
  else else_action
until exit is specified by an event device

```

Normally the action is generation of a prompt for the next parameter to be supplied by the user as part of a command specification dialogue. When a command is fully specified, the action modifies the data structure and then, most likely, the picture as well.

This general schema allows the application program to wait for more than one event-generating device at a time; for example, while typing an input string, the program could also wait on CANCEL and HELP buttons. This generality necessitates

*We use the term *event* here in the sense of a user-generated interrupt that causes a change of state in the system, i.e., from wait to action (and then back to wait). Thus the generation by the application program of a new picture, of feedback or of prompts is not considered an event.

somewhat complex event-decoding logic and status-saving (Exercise 2.12). SGP therefore uses a simpler schema in which the application program can wait on only a single event-generating device at a time. When the application program gains control after the event has taken place, it does a **case** on the *event data* returned by that event-generating device which indicates the exact choice the user made. This simpler schema looks the same, but the **case** is on event data, not the event device:

```

initialize;
repeat
  WAIT for user-generated event;
  if event_data = case 1 or case 2 or . . . then    {"legal" action}
    case event_data of                             {perform appropriate action}
      .
      .
      .
    end      {case}
until exit

```

Even in the more complex situation, the **case** is actually on the event data returned by the event-causing device, not just the device as shown above.

The event-generating devices supported by SGP are the logical button (typically implemented with a programmable function keyboard), the alphanumeric keyboard, and the pick device (usually implemented with a light pen or a data tablet with comparator). Each of these devices is activated by the programmer by having SGP execute the particular *WAIT* procedure for that device; an SGP-defined echo lets the user confirm that the input is being accepted and processed. If the user activates an event-generating device that is not being *WAIT*ed on, the absence of the corresponding echo indicates that the device's input is being ignored.

The *WAIT_BUTTON* procedure is typically used to choose a program option as a reply to a prompt or to specify a global change in mode of operation of the application program. The invocation of the procedure

WAIT_BUTTON(time, button_name)

causes the button device to be enabled and the application program to wait until either time-out occurs (if the user has not pressed a button during the number of seconds specified by the parameter *time*) or the user presses a key.* SGP learns from the hardware which key was pressed (this is the associated event data), returns the key's identifying number in the parameter *button_name*, and lets the application program resume execution at the statement following the *WAIT_BUTTON*. Time-out causes an illegal key number of 0 to be returned. An appropriate echo for button selection might be to turn on or blink the light under the button selected.

Similarly,

WAIT_PICK(time, segment_name)

*This description assumes a physical button keyboard, but the logic is identical if the logical button device is being simulated by another physical device, as discussed in Chapter 5.

will either time-out or return as event data the name of the segment whose primitive is identified on the screen by the user. The appropriate echo might be to blink the selected primitive or the entire segment. *WAIT_PICK* is used, for example, to indicate a segment to be deleted; as we will see, it can also be used in identifying choices as an alternative to buttons. The mechanisms used by SGP to correlate the primitive picked by the user to the name of the segment containing the primitive are discussed in the following two chapters.

The procedure

WAIT_KEYBOARD(time, text, length)

allows the user to type characters until an event is caused by hitting the carriage-return key (or its equivalent), and then returns as event data the string (minus the carriage return) in parameter *text* and its length in parameter *length*. An appropriate echo might be to display the typed characters at successive positions of a blinking cursor in a standard *type-in area* on the screen.

2.13 MAINLINE USING BUTTONS FOR FUNCTION INVOCATION

Example 2.3 To begin the demonstration of interactive programming, we first show a preliminary (partially pseudocoded) version of the mainline program for the symbol placement application described earlier. In this version of the mainline the logical button device is used to allow the user to specify the function to be invoked next.

The user of the symbol placement program may choose any of the following functions:

- Specify a new title for the picture;
- Add a symbol to the data structure and the picture;
- Delete a symbol from the data structure and the picture;
- Change the window for the picture; and
- Terminate execution.

In this version of the mainline program, a unique button on the button device is assigned to each of these functions. When a button event occurs, the program performs the function associated with the particular button activated.

The application program procedure *SETUP*, which is called by the mainline, performs all necessary initialization actions. In particular, it reads from secondary storage the data structure information defining the room whose furniture is to be arranged. In addition to reading values into the *symbols* array and *title* variable defined in Section 2.11, *SETUP* reads information describing the size and shape of the room and locations of doorways and windows. This information is used by the *DISPLAY_ROOM* procedure (not elaborated further here) to create and display the segment defining the room boundary seen in Fig. 2.3. Finally, *SETUP* initializes the four global variables *room_min_x*, *room_max_x*, *room_min_y*, and *room_max_y*, so that they define a window in world coordinates that just surrounds the limits of

the room; this is the largest window that can be used to display the room. The application program procedure *CLEANUP* is the opposite of *SETUP*. It performs all necessary termination processing, including writing the updated data structure values to secondary storage.

The interactive portion of the mainline *WAITs* for a button to be pushed and then invokes the lower-level procedure corresponding to the button. When the user is finished, the function returns from the *case* to the *WAIT*. The mainline therefore shows only the top-level, *dispatcher table* portion of the user dialogue, not the more interesting interactions which accomplish each of the functions. If no event occurs within the period specified by the symbolic constant *wait_time*, say five minutes, the *WAIT* times out and the user is prodded with a prompt or by highlighting the previous prompt—ringing an audible alarm or using some other attention-getting device. The *SCREEN_FEEDBACK* procedure should delete its message from the screen after a short wait and then return to exit in the *case* statement (see Exercise 2.15).

```
{Mainline for Symbol Placement Program—Version 1, using buttons for interaction}
program LAYOUT(input, output);
declare variables;
begin
  {initialize}
  SETUP;
  WINDOW(room_min_x, room_max_x, room_min_y, room_max_y);
  VIEWPORT_2(0.0, 0.8, 0.0, 0.8);    {leave room for prompts}
  {generate initial display}
  DISPLAY_D_STR;
  SCREEN_FEEDBACK('press a button to select a function');

  {main event-driven loop}
  main_done := false;
  repeat
    {wait for user-generated event}
    WAIT_BUTTON(wait_time, button_number);
    if (button_number >= 0) and (button_number <= 5) then    {"legal" button}
      case button_number of
        0: SCREEN_FEEDBACK('waiting for you to select a function');
        1: NEW_TITLE;
        2: ADD_SYMBOL;
        3: DELETE_SYMBOL;
        4: CHANGE_VIEW;
        5: main_done := true    {EXIT button}
      end {case}
    else SCREEN_FEEDBACK('illegal selection, try again')
  until main_done;
  CLEANUP    {application procedure to terminate}
end.
```

2.14 USING A MENU FOR FUNCTION INVOCATION

2.14.1 Light Buttons

Using buttons to specify functions has some disadvantages. It can be inconvenient and distracting for the user to shift attention from the screen to the box in which the buttons are usually located. Secondly, most function keyboards have no means for labeling the buttons under program control. Although it is possible to show a brief summary of the use of each button on a portion of the screen, the user's attention is still divided.

A technique that many users prefer is to display a *function menu of light buttons* on the screen from which the desired function may be indicated with the pick device. A typical menu is composed of the list of text strings and/or graphical symbols that represent the functions available. Figure 2.13 illustrates a typical text string menu. The disadvantage of using light buttons (they take up some valuable screen space) is usually outweighed by the ability to display labeled choices and to change quickly from one menu to another.

For picking purposes, each light button string is defined by the routine *CREATE_FUNC_MENU* (not shown here) in its own uniquely named segment and is displayed in its own viewport. Each segment contains an absolute *MOVE* followed by a *TEXT* primitive, and is created during the *SETUP* initialization phase. The event-driven loop is also altered slightly; instead of

```
WAIT_BUTTON(wait_time, button_number);  
case button_number of
```

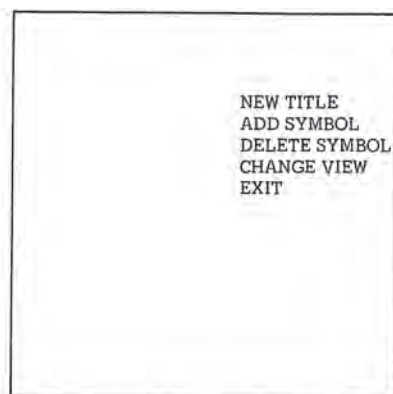


Fig. 2.13 Text string menu generated by *CREATE_FUNC_MENU* procedure.

returns to *ADD_SYMBOL* via its own DONE or CANCEL, the user can either pick another symbol for *PLACE_SYMBOL* or choose DONE or CANCEL (to return to the mainline). The CANCEL option in *ADD_SYMBOL* allows the user to undo a prior placement before returning to the mainline. If *PLACE_SYMBOL* encounters a condition of no more room in the data structure, it sets the exit flag *add_done* so that both *PLACE_SYMBOL* and *ADD_SYMBOL* are exited. These various options, including multiple placements, are summarized in the state transition graph of Fig. 2.15.

Only the most recently placed symbol may be canceled with the CANCEL option. To implement CANCEL, *ADD_SYMBOL* saves the segment name of the most recently placed symbol (i.e., the symbol's location in its data structure array) in the variable *symb_addr*. When the user invokes CANCEL, this segment and the corresponding entry in the data structure are deleted. The *symb_addr* flag is initialized to 0, indicating that no cancel is yet possible, and is set by the *PLACE_SYMBOL* procedure when a symbol is placed. Also, CANCEL causes *symb_addr* to be reset to 0 in both *PLACE_SYMBOL* and *ADD_SYMBOL* since no second cancellation is possible in the other routine. Procedure *ADD_SYMBOL* is shown on p. 72.

2.16.3 Positioning with the Locator Device

The *PLACE_SYMBOL* procedure in Section 2.16.5 below does the work of creating the new symbol and adding it to the data structure. However, the user must specify the position and orientation of the new symbol: the locator device is used to position the new symbol and the valuator device is used to specify its orientation.

The *locator* device allows the user to specify a position on the screen, typically in order to place symbols at a desired location or to "draw" one or more primitives by indicating (end)points to the application program. The locator device does *not* cause events and is called a *sampled device* because the application program must explicitly ask for (i.e., *sample*) the current (*x*, *y*) values of the locator. Some typical physical devices that can be easily used as locators include data tablets, joysticks, and thumbwheel-driven cross-hair cursors. They return two coordinates in the NDC range of 0.0 to 1.0 which can be echoed on the screen by a cross-hair or a blinking cursor at the corresponding screen position. The data associated with the locator can be accessed by the following call:

READ_LOCATOR(*x*, *y*)

In most cases the coordinates returned from the locator are of little immediate use because they are given in the normalized device coordinate system and reflect where on the screen the cursor is physically located. However, the program typically views the picture on the screen as a reflection of the model's world coordinate system, and therefore must know where the cursor is in that space. This is important if, for example, the cursor is being used to specify endpoints of lines being constructed by the user, or in the case of our *LAYOUT* program, to specify where a particular furniture symbol is to be located in the (world coordinate) office.

```

procedure ADD_SYMBOL;
{add symbols to the picture and data structure}
var
  i: integer;
  symb_addr, segment name: integer;
  add_done: boolean;
begin
  symb_addr := 0;
  SET_VISIBILITY(done, on);
  SET_VISIBILITY(cancel, on);

  add_done := false;
  repeat
    for i := mchair to mfloorplant do    {symbol menu and prompt visible}
      SET_VISIBILITY(i, on);
      SET_VISIBILITY(add_prompt, on);
      WAIT_PICK(wait_time, segment_name);
      for i := mchair to mfloorplant do
        {since choice picked, make symbol menu and prompts invisible}
        SET_VISIBILITY(i, off);
      SET_VISIBILITY(add_prompt, off);
      if ((segment_name >= help)    and (segment_name <= cancel)) or
        ((segment_name >= mchair) and (segment_name <= mfloorplant)) then
        case segment_name of    {valid menu pick}
          0: SCREEN_FEEDBACK('waiting for selection');
             mchair, mdesk, mbookcase, mlowdivider, mpartition, mfloorplant:
             {picked a symbol from menu so place a copy; its address is returned in}
             symb_addr
             PLACE_SYMBOL(segment_name, symb_addr, add_done);
             help: HELP_USER(add_help);
             done: add_done := true; {DONE}
             cancel: if (symb_addr > 0) then    {CANCEL only if symb_addr
               flag on, i.e., no prior CANCEL in ADD_SYMBOL or
               PLACE_SYMBOL}
               begin
                 {delete segment, then symbol}
                 DELETE_SEGMENT(symb_addr);
                 symbols[symb_addr].code := 0;
                 symb_addr := 0 {set flag off to prevent second CANCEL}
               end {CANCEL}
             else SCREEN_FEEDBACK('nothing to cancel')
           end {case}
        else SCREEN_FEEDBACK('illegal selection, try again')
      until add_done;
      SET_VISIBILITY(done, off);
      SET_VISIBILITY(cancel, off)
    end; {ADD_SYMBOL}

```

To facilitate this *back-mapping* from NDC space to world space, SGP contains the subroutine *INVERSE_2* which transforms a position in normalized device coordinates into a position in world coordinates. The transformation is the inverse of the current viewing operation, that is, the inverse of the mapping that takes the world coordinate window into the NDC viewport (discussed in Chapter 4). The statement below is an invocation of the procedure:

```
INVERSE_2(ndc_x, ndc_y, world_x, world_y)
```

This procedure must be used carefully because, although a given display on the screen can be generated as a collection of individual images, each specified via its own window-viewport viewing operation, only the current (most recently specified) viewing operation can be used to compute the inverse mapping. In other words, SGP does not remember for each segment what viewing operation was in effect when it was created. Since locator cursor positions are typically used only within the main drawing area, we can program so that this area is the last one drawn; this allows SGP to use the last-specified viewing operation to calculate the inverse.* If multiple drawing areas with multiple viewing operations are required, the application program would have to remember which viewing operation belongs to which viewport and segment combination and would then compute its own inverses.

In our sample program, the mainline's *SETUP* (and some of the procedures it calls) will change the viewport for each menu item; after that, the viewport will be set to the main picture-building area of the screen and left that way to allow *INVERSE_2* to be used for all subsequent operations.

2.16.4 The Valuator Device

The *valuator* logical device allows a user to specify to the application program a single floating-point value between 0.0 and 1.0. Valuator, like locators, are sampled devices and do not cause events. The physical device most often utilized as a logical valuator device is a control dial. A control dial is simply a potentiometer similar to that used as a volume control on a radio or television set. The analog output of the dial is converted to a digital value by hardware, which may be echoed in a control area on the screen or, more likely, by a pointer on a scale attached to the device itself. The *READ_VALUATOR* procedure must be called in order for the application program to obtain the value of a valuator. The valuator specified by the first parameter of this procedure is sampled and the value is returned in the variable specified as the second parameter. The statement

```
READ_VALUATOR(1, value)
```

sets the variable *value* to the current value of valuator 1.

*Note that the locator is not constrained to return values inside a particular viewport; therefore, if the locator returns a point not inside the current viewport, and this point is passed on to *INVERSE_2*, a point lying outside of the current window will be returned as the inverse point.

5

Interaction Devices and Techniques

We have already introduced, in the preceding chapters, a number of interaction devices and techniques. The devices include the light pen, data tablet with stylus, and alphanumeric and function keyboards. Techniques for using the devices include selecting commands from a menu with the light pen or tablet and positioning symbols with a tablet. In this chapter we describe more (but not all) of the available devices and techniques and touch on their relative merits.

Our objective is to impart an understanding of how the devices work and how they can be used. The devices and techniques are the building blocks from which an interactive dialogue is constructed. In Chapter 6 we consider methodology and guidelines to help in selecting the set of devices and techniques to be used in a specific application and in integrating these building blocks into a usable dialogue.

5.1 LOGICAL CLASSES OF DEVICES AND TECHNIQUES

There is a large variety of interaction devices and techniques. Fortunately, we can impose some organization on them and thereby structure our study and use of the techniques. The organization employed is that of *logical devices*, introduced in Chapter 2 as part of SGP. There are five basic logical devices: the *locator*, to indicate a position and/or orientation; the *pick*, to select a displayed entity; the *valuator*, to input a single value in the space of real numbers; the *keyboard*, to input a character string; and the *button*, to select from a set of possible alternative actions or choices.

Each logical device has a natural prototype in a specific physical device or class of devices. In SGP, for example, the pick device was discussed in terms of its natural prototype, the lightpen. Other natural prototypes are: locator—tablet; valuator—potentiometer; keyboard—alphanumeric keyboard; button—programmed

function keyboard. However, any of these logical devices can be simulated by any input device: many of the interaction techniques we will discuss are such simulations.

The concept of logical input devices is rather like that of logical files in an operating system. A sequential input file may be implemented physically by means of a card reader, a magnetic tape drive, a disk drive, or a terminal keyboard. The application programmer doesn't care which one it is—the operating system makes them all “look alike” functionally, despite their physical differences.

The essence of the logical-device idea is found in a 1968 paper by Newman [NEWM68b]. In 1971 the *general-purpose graphic system* (GPGS) subroutine package [VAND77] implemented interaction by using logical devices and showed the utility of simulating input devices, such as picks, on simple display devices lacking physical pick mechanisms. A series of articles by Cotton [COTT72], Wallace and Foley [FOLE74] and Wallace [WALL76] elaborates on the concept. In addition, several higher-level devices (such as a *sketch* device) for specifying a sequence of positions are described in [WALL76]. Logical devices are found not only in GPGS but also in the proposed *Core System* [GSPC79] and in the *ARPA network graphics protocol* [SPRO74].

5.2 PHYSICAL INTERACTION DEVICES

In this section we describe how some interaction devices actually work. Many devices are in use, so we focus on those which are more common or more innovative. The devices are grouped by the logical-device category which matches the inherent physical operation of each device—the logical device which the physical device implements with a minimum of support software. Realize, however, that ease of implementation may not be equivalent to ease of use. We are thus not proposing that the physical devices are best used to implement the logical device under which they are here categorized. Throughout this chapter the reader will find comparisons and contrasts among different logical device implementations, meant to help in the actual process of selecting the appropriate interaction technique for implementing a logical device.

5.2.1 Locators

The most commonly used locator, the *tablet*, is a flat surface over which a *stylus* (like a pencil) or *hand cursor* is moved; the position of the hand cursor or stylus is available to the computer. A tablet with a stylus is shown in Fig. 1.14; a back-lit tablet with a hand cursor is shown in Fig. 5.1. The stylus typically incorporates a pressure-sensitive switch which closes when the user pushes down on the stylus, thus indicating to the application program that the stylus is at a position of interest. Most hand cursors have several buttons which can be operated by the user to input commands. The stylus switch and the hand cursor buttons are really separate logical button devices, physically but not necessarily logically integrated into the locator device.

function keyboard. However, any of these logical devices can be simulated by any input device: many of the interaction techniques we will discuss are such simulations.

The concept of logical input devices is rather like that of logical files in an operating system. A sequential input file may be implemented physically by means of a card reader, a magnetic tape drive, a disk drive, or a terminal keyboard. The application programmer doesn't care which one it is—the operating system makes them all “look alike” functionally, despite their physical differences.

The essence of the logical-device idea is found in a 1968 paper by Newman [NEWM68b]. In 1971 the *general-purpose graphic system* (GPGS) subroutine package [VAND77] implemented interaction by using logical devices and showed the utility of simulating input devices, such as picks, on simple display devices lacking physical pick mechanisms. A series of articles by Cotton [COTT72], Wallace and Foley [FOLE74] and Wallace [WALL76] elaborates on the concept. In addition, several higher-level devices (such as a *sketch* device) for specifying a sequence of positions are described in [WALL76]. Logical devices are found not only in GPGS but also in the proposed *Core System* [GSPC79] and in the *ARPA network graphics protocol* [SPRO74].

5.2 PHYSICAL INTERACTION DEVICES

In this section we describe how some interaction devices actually work. Many devices are in use, so we focus on those which are more common or more innovative. The devices are grouped by the logical-device category which matches the inherent physical operation of each device—the logical device which the physical device implements with a minimum of support software. Realize, however, that ease of implementation may not be equivalent to ease of use. We are thus not proposing that the physical devices are best used to implement the logical device under which they are here categorized. Throughout this chapter the reader will find comparisons and contrasts among different logical device implementations, meant to help in the actual process of selecting the appropriate interaction technique for implementing a logical device.

5.2.1 Locators

The most commonly used locator, the *tablet*, is a flat surface over which a *stylus* (like a pencil) or *hand cursor* is moved; the position of the hand cursor or stylus is available to the computer. A tablet with a stylus is shown in Fig. 1.14; a back-lit tablet with a hand cursor is shown in Fig. 5.1. The stylus typically incorporates a pressure-sensitive switch which closes when the user pushes down on the stylus, thus indicating to the application program that the stylus is at a position of interest. Most hand cursors have several buttons which can be operated by the user to input commands. The stylus switch and the hand cursor buttons are really separate logical button devices, physically but not necessarily logically integrated into the locator device.

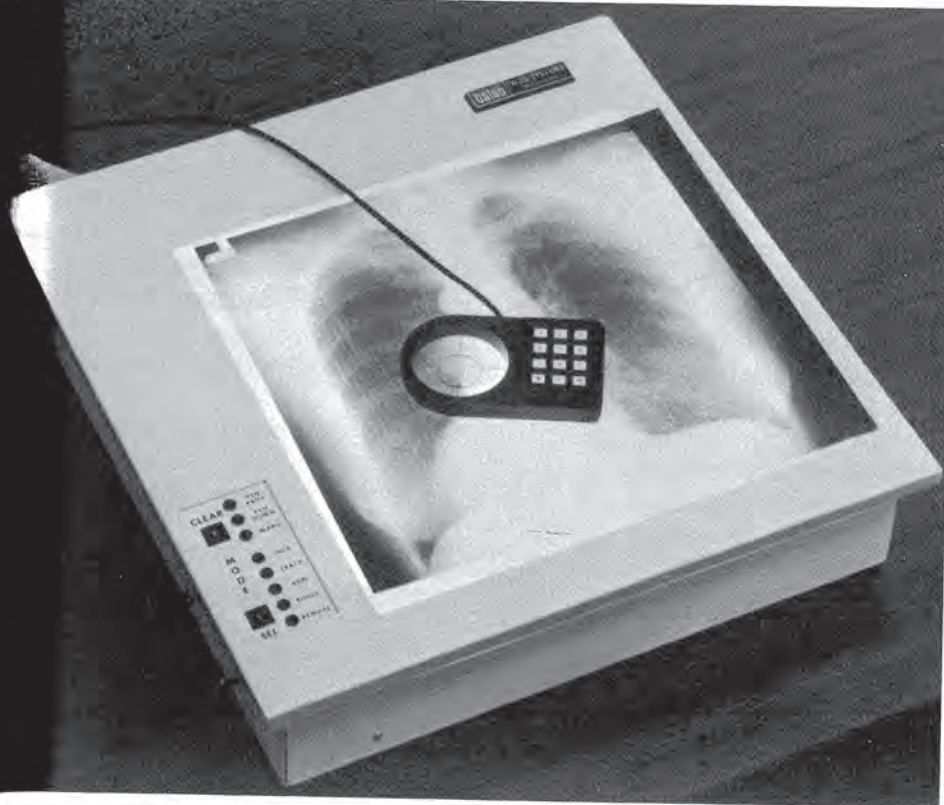


Fig. 5.1 Back-lit tablet with 12-button cursor (courtesy Talos Systems, Inc.).

Most tablets use an electrical sensing mechanism to measure the stylus or hand cursor position. In one such arrangement, a grid of wires is embedded in the tablet surface. Electromagnetic coupling between electrical signals in the grid and in the stylus or cursor induces an electrical signal in a wire coil in the stylus or cursor. The strength of the electromagnetic coupling can be used to determine roughly how far the stylus or cursor is from the tablet ("far", "near", "on"). When the answer is "near" or "on", a *screen cursor* is usually displayed (by the display hardware or graphics package) to track the position being read from the tablet and thus to provide visual feedback to the user.

Information is typically obtained from a tablet controller in one of three ways: on demand, when the computer makes a request; every t units of time; and each time the hand cursor or stylus is moved more than some distance d . The time-interval sampling is helpful if a screen cursor is being displayed to reflect the position of the tablet cursor or stylus. The distance-interval sampling is appropriate for digitizing drawings in order to avoid recording an excessive number of points.

Relevant parameters of tablets and other locator devices are their resolution, linearity, repeatability, and size or range. These parameters are particularly crucial for digitizing maps and drawings, but are of less concern when the device is used only to position a screen cursor, since the user has the feedback of the screen cursor position to guide his hand movements. Tablets of at least 48×72 inches with accuracy of 200 points per inch are commercially available.

Another locator is the *mouse* shown in Fig. 5.2. It is a hand-held device with rollers on its base. Moving the mouse across a flat surface causes the rollers to turn, and potentiometers coupled to one roller sense the relative movements in two orthogonal directions. The motion is converted to digital values and is used to determine the direction and magnitude of the mouse's movement. The mouse can be picked up, moved, and then put down without any change in the apparent position of its screen cursor. Thus the mouse reports relative movement, while the tablet reports absolute positions. The computer must maintain a "current mouse position" which is incremented or decremented by mouse movements. The mouse, like most hand cursors, usually has several push buttons which can be used to input commands or other information and hence serve as logical buttons; the pick function is thus easily implemented.

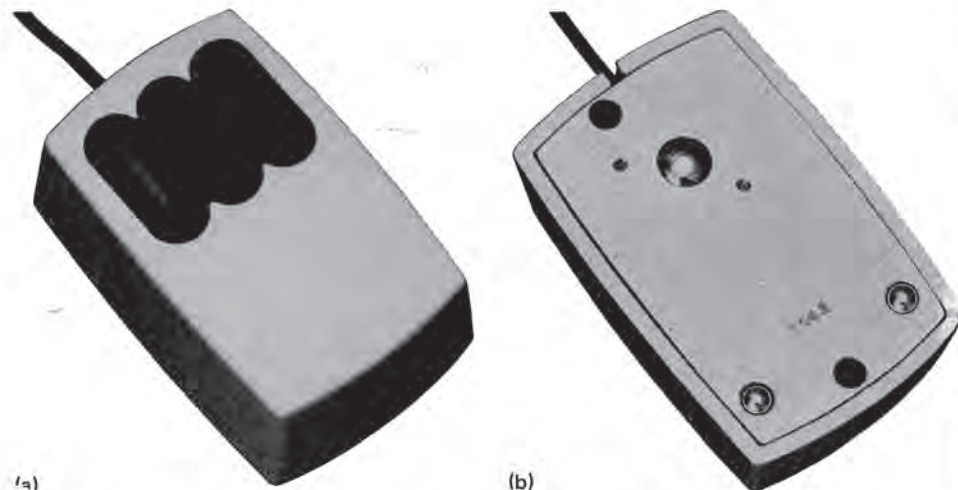


Fig. 5.2 The mouse: (a) top view; (b) bottom view (courtesy Xerox Corporation).

While the accuracy of the mouse is comparable to that of a tablet, very fine and precise sketching is more difficult than with a tablet and stylus because fine fingertip motor control is more difficult. On the other hand, a mouse requires only a small table area to use and is inexpensive to manufacture.



Fig. 5.3 Trackball (courtesy Ramtek Corporation).

Trackballs, sometimes called *crystal balls*, are another form of locator (Fig. 5.3). The ball's motion turns potentiometers, whose output is converted into digital form. The ball rotates freely within its mount, and is typically moved by drawing the palm of one's hand across it. Large and rapid position changes are difficult to make with a trackball.

The *joystick* is simply a stick which can be moved left or right, forward or backward (Fig. 5.4). Again, potentiometers sense the movements. Very often, sets of springs are used to return the joystick to its home center position. Some joysticks, including the one pictured, have a third degree of freedom: the stick can be twisted clockwise and counterclockwise. The *isometric joystick*, shown in Fig. 5.5, is rigid: strain gauges on the shaft measure slight deflections caused by force applied to the shaft.

It is difficult to use a joystick to control the absolute position of a screen cursor directly, because a slight hand movement is amplified five or ten times in the position of the screen cursor. This makes the screen cursor's movements quite jerky and doesn't allow quick and accurate fine positioning. Thus the joystick is often used to control the *velocity* of the screen cursor's movement rather than its absolute position. This means that the current position of the screen cursor is changed over time at rates determined by the joystick. It is important that a small *dead zone* of zero



Fig. 5.4 Joystick with rotating shaft to provide third axis (courtesy Vector General).



Fig. 5.5 Rigid joystick (courtesy Measurement Systems, Inc.).

velocity be established to allow for drift in the joystick's home center position. To further enhance ease of use, the relation of cursor velocity to joystick displacement is often made nonlinear, as shown in Fig. 5.6. This velocity curve, along with the spring return on the joystick, makes positioning fairly fast, but not as fast as with a tablet or mouse.

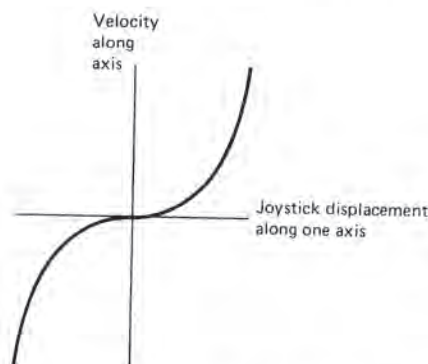


Fig. 5.6 Nonlinear velocity control.

The *joystick*, a variant of the joystick, can be moved in any of eight directions: up, down, left, right, and in the four diagonal directions. The switch hence has nine states. In each of the eight "on" states, the position of the screen cursor is changed at a *constant* rate, in the appropriate direction. The difficulty with this is in moving the screen cursor over large distances.

The joystick can also be used conveniently as a 3D orientation device (logical locators can provide both position and orientation), especially if the shaft rotates with a third degree of freedom. The three values from the joystick can be used to control the rate of rotation about each of the three axes. If the joystick does not have spring return to zero, its values can be used as absolute rotations rather than as rates of rotation.

These locator devices are manipulated with the user's hands, while the screen cursor is watched with the eyes. The *touch panel*, on the other hand, is a locator device which allows the user to give full attention to the screen and to indicate positions on it directly rather than by driving a screen cursor to the desired location. The transparent panel is mounted across the face of the CRT: when the user's finger touches the panel, the position touched is detected.

Several different technologies are used for touch panels. Low-resolution panels (from 10 to 50 resolvable positions in each direction) use a series of light sources (light-emitting diodes) and light sensors (photodiodes or phototransistors) to form a grid over the display area. Touching the screen breaks one or two vertical and horizontal light beams, thereby revealing the finger's position. If two parallel beams are broken, the finger is presumed to be centered between them; if one is broken, the finger is presumed to be on the beam. One high-resolution panel (about 500 resolvable positions in each direction) uses sonar-style ranging. Bursts of high-frequency shock waves traveling alternately horizontally and vertically are induced in a flat glass plate. The touch of a finger on the glass causes part of the wave to be reflected back to its source. The distance to the finger can be calculated from the time interval between emission of the wave burst and its arrival back at the source. Another high-resolution panel uses two slightly separated layers of transparent material, one

coated with a thin substrate of conducting material; the other, resistive. Finger-tip pressure forces the layers to touch, and the voltage drop across the resistive substrate is measured and used to calculate the coordinates of the touched position.

The *sonic tablet* uses sound to couple the stylus to strip microphones along two sides of the tablet. An electrical spark is generated at the tip of the stylus. The time delay between when the spark occurs and when its sound arrives at each microphone is proportional to the distance of the stylus from each microphone, as suggested by Fig. 5.7.

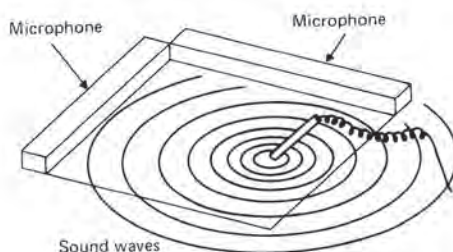


Fig. 5.7 2D sonic tablet.

All the devices we have discussed are 2D locators. It is often convenient to have 3D devices as well, even though 2D devices can be used in sequence to specify 3D locations. The sound coupling concept can be easily extended to create a 3D *sonic pen*. In one approach, three orthogonal strip microphones are used, as illustrated in Fig. 5.8. Each microphone fixes the position of the stylus on a cylinder of known radius (determined by the "time of flight" of the sound to the microphone) centered on the microphone. The intersection of the three cylinders determines the position of the stylus. The equations of the three cylinders are:

$$x^2 + y^2 = r_z^2, \quad y^2 + z^2 = r_x^2, \quad z^2 + x^2 = r_y^2, \quad (5.1)$$

where r_x , r_y , and r_z are the measured radii. Another way to use strip microphones for 3D is to arrange four of them in a square.

An earlier sound-based locator, the *Lincoln Wand* (so named because it was developed at Lincoln Labs), had four point microphones at the corners of the display

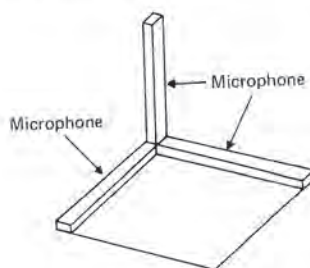


Fig. 5.8 3D sonic tablet.

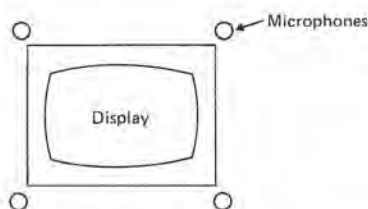


Fig. 5.9 Microphone arrangement for the Lincoln Wand.

area (Fig. 5.9). Again, time of flight is measured, but now the times determine the radii of four spheres. Each sphere is centered on one microphone. (Actually, only three spheres are needed to determine a position; the fourth sphere can be used for error checking or in case the sound path to one microphone is blocked.) The spherical equations are all of the form

$$(x - x_i)^2 + (y - y_i)^2 + (z - z_i)^2 = r_i^2, \quad i = 1, 2, 3, 4. \quad (5.2)$$

The (x_i, y_i, z_i) are the positions of the four microphones. Solving the equations is left as an exercise for the reader.

Several mechanically coupled 3D locator devices have been developed. One, the *Noll Box* (after its developer, Michael Noll), is just a large box with slide mechanisms allowing the knob to be moved anywhere in a 12-inch cube (Fig. 5.10). The vertical shaft is counterbalanced so that it maintains any established position, and long slide potentiometers are used to record the position along each axis. Another mechanically coupled 3D locator is described in [GEYE75].

These 3D locators all work in relatively small volumes—several cubic feet at most. One of the dreams of computer graphics researchers has been to sense the 3D motion of the user's hands and fingers, so that they can be used to define positions and even shapes in 3D. This has been accomplished through the use of optical coupling between active light sources on the user (light-emitting diodes) and sensors mounted high in the corners of a small room.

Two different sensor mechanisms have been developed [BURT74, FUCH77a]. The sensor is organized to provide the parameters of a plane on which the light source is located. Three sensors defining three planes are therefore sufficient to determine the position of the light source; again, a fourth sensor can be used for error-checking and for cases when one of the light sources is obscured from a sensor. The equations for the planes are of the form:

$$a_{i1}x + a_{i2}y + a_{i3}z + a_{i4} = 0, \quad i = 1, 2, 3, 4. \quad (5.3)$$

Mechanical, sound, and light coupling to determine 3D positions are subject to noise and obstructions in the work area. The Polhemus Navigation Sciences ROPAMS system generates electromagnetic fields which are detected by a small sensor mounted on a hand or other part of the body. The position and orientation of the sensor can be determined very precisely. This technology is still too expensive for all but experimental use.

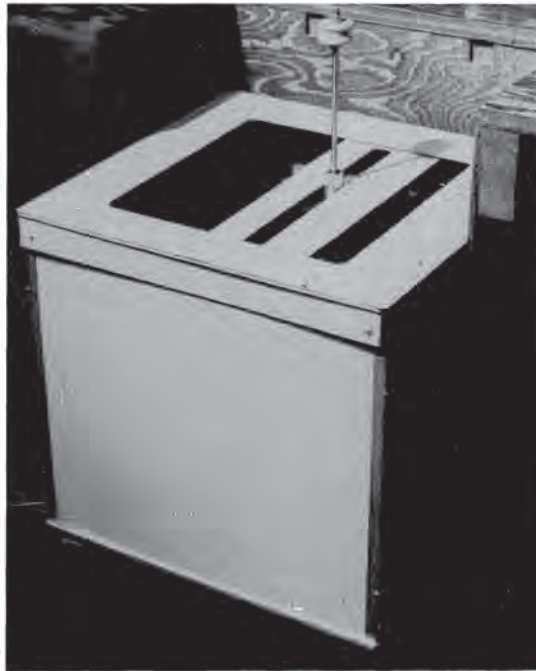


Fig. 5.10 3D locator device.

5.2.2 Picks

The only device which is inherently a pick device is the light pen. As we saw in Chapter 3, the pen's output is usually connected to the DPU's control logic, so that the DPU stops executing commands when the pen sees light. The DPU stops with its instruction counter containing the address of the instruction following the one that drew the detected output primitive.*

What will the DPU's X and Y position registers contain if the light-pen "hit" occurred while a vector was being drawn? With an analog vector generator, the end-points of the vector will be in the registers, because points along the vector are never in digital form. With a digital vector generator (increasingly common with contemporary vector displays), the coordinates of a point on the vector a little past the point seen by the light pen will be in the register, and can be made available to the CPU. These coordinates can be used, for instance, in creating a connecting line somewhere in the middle of an existing line.

*In some high-performance displays, the time for the light pen to halt the DPU may be longer than the time required to draw one or more short vectors or characters, so that the address counter could point to an element further down in the DPU program. Also, the DPU may be pipelined, beginning the processing of one instruction before the preceding one is completed. To get precise pen detects, the graphics package may have to insert graphic "no-op" commands after each group of elements which is to be uniquely identifiable.

The light p
locator on a r
ake the x and
associated comp

Figure 5.11
ht pen and th
n be controll
justing the le
is shown here
mechanical shut
d dispense wi

g. 5.11 Seve

The light pen, which implements a logical pick on a vector display, implements a locator on a raster display. When a pixel is detected, the image display system can make the x and y coordinates of the raster scan's current position available to the associated computer.

Figure 5.11 shows several light pens, while Fig. 5.12 shows a cross-section of a light pen and the fundamentals of its associated electronics. The pen's field of view can be controlled by putting different sizes of apertures in front of the lens or by adjusting the lens focus. The switch can be used to enable and disable the light pen. It is shown here as controlling an electrical circuit, but on some pens it is linked to a mechanical shutter. Also, some pens place the photomultiplier tube in the pen barrel and dispense with the fiber optics bundle.

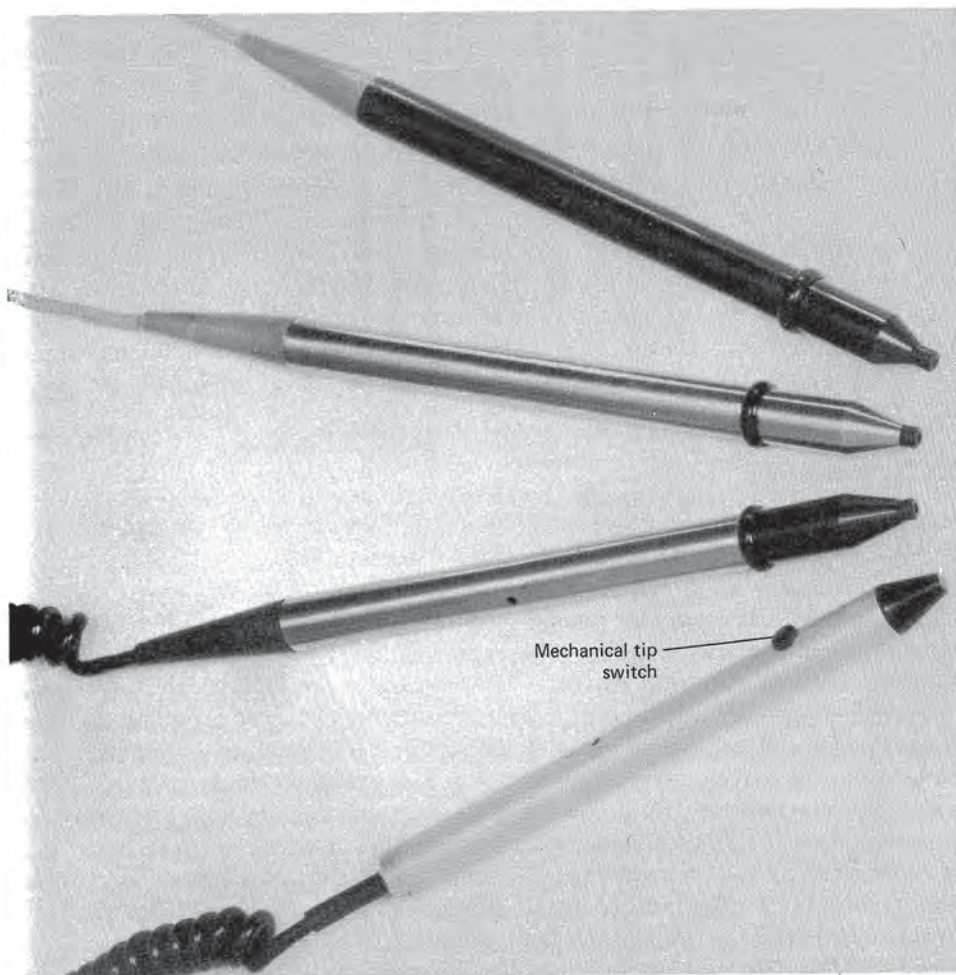


Fig. 5.11 Several light pens. One pen has mechanical tip switch; the others, capacitive.

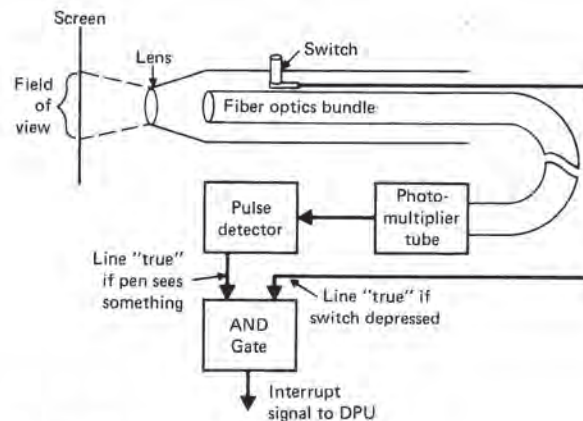


Fig. 5.12 Diagram of light pen.

Light pens were developed very early in the history of interactive computer graphics, originally for use in the SAGE air defense system to point at targets on a radar screen. Unless properly adjusted, light pens may detect false targets, such as fluorescent lights or nearby primitives (e.g., adjacent characters) and fail to detect intended targets. When used over a period of several hours, a pen can be tiring to the user, who must pick it up, point it, and set it down for each use. As a consequence, more and more systems employ the tablet or mouse to simulate the picking capability of a light pen. As raster graphics grows, the popularity of light pens can be expected to decrease even further.

5.2.3 Valuator

Most valuator, devices which provide scalar values, are based on the potentiometer. The volume, balance, and tone controls on a stereo set, for instance, are potentiometers.

Most valuator are rotary potentiometers (dials), typically mounted in a group of eight or ten, as pictured in Fig. 1.14. Rotary potentiometers are especially appropriate for controlling object rotation. Slide potentiometers, in which linear movement replaces rotation, are appropriate for specifying values with no angular interpretation. The current setting of a slide potentiometer is much easier for the user to deduce than that of a rotary potentiometer; unfortunately, most graphics system manufacturers offer only rotary potentiometers. Availability of both types of potentiometers can help users remember what functions are associated with each valuator. When designing a system involving potentiometers, one must be sure to adopt a consistent interpretation for increasing and decreasing values: for instance, clockwise or upward movements normally increase a value.

An analog-to-digital converter and a power supply can determine the potentiometer's position by measuring the voltage V_1 (Fig. 5.13). The voltage is propor-

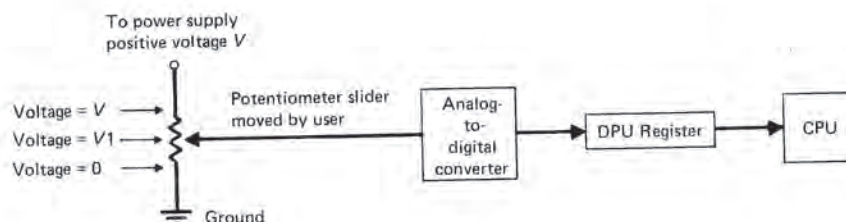


Fig. 5.13 Potentiometer interfaced to DPU.

proportional to the amount of the shaft rotation about the corresponding axis. If the converter is properly adjusted, zero rotation will correspond to a digital reading of all zeros, while full rotation will correspond to all ones. We assume that the converted values are placed into DPU device registers which can then be read by the CPU.

For some applications, it is convenient to have the graphics package poll the valuator to see if it has been changed more than some small amount, and then to generate an event which is placed on the graphics package event queue. This means the application program need not monitor the valuator via a polling loop to see if it has changed. Both programmer effort and CPU time are saved by supporting polling at the lower, more efficient level. Notice too that the same technique could be applied to locator devices.

5.2.4 Keyboards

The alphanumeric keyboard is our prototype (and only) text device. Several different technologies are used to detect a key depression: some of these are mechanical contact closure, change in capacitance, and magnetic coupling. The important functional characteristic of a keyboard device is that it creates a code (ASCII, EBCDIC, etc.) uniquely corresponding to a depressed key.

There are many factors which make one keyboard preferable to another: inter-key spacing, slope of the keyboard, shape of the key caps, pressure needed to depress a key, and a feeling of contact when a key is fully depressed. Many keyboard design issues are summarized in [KLEM71] and [KROE72]. Other considerations, dealing not with hardware but with software design, are making frequently used delimiter or correction keys easily reachable *without* the need to simultaneously depress the control or shift keys, and separating dangerous keys (such as line delete) from other frequently used keys (such as return).

5.2.5 Buttons

The *programmed function keyboard* (PFK) is the most common button device. It is sometimes built as a separate unit, but more often the buttons are integrated with a keyboard. Other button devices are the keys found on many tablet cursors and on

the "mouse." Buttons are generally used to enter commands or menu options to a graphics program. Dedicated-purpose systems can use buttons with permanent labels. To allow changing or "soft" labels, some PFKs can be used with coded overlays on which command names are printed. The presence or absence of tabs on the edge of the overlay is sensed, so the system knows which overlay is in place. The operator inserts the overlay containing the desired command and then depresses the appropriate button. Knowlton's keyboard, which uses mirrors to superimpose labels from a CRT on top of an unlabeled keyboard, provides a more general sort of soft label capability [KNOW75]. Another way to have soft labels is by having the buttons at the edge of the view surface, allowing labels to be displayed.

How does a button device differ from a text device? The text device provides input from buttons which are prelabeled. The button device keys have no predefined character meanings, and the number of keys may vary from one to many. Further, some button devices report button releases as well as button depresses. This makes it easy to start an activity (such as rotation of a displayed object) when a button is depressed and then to terminate the activity when the button is released. One can consider the text device to be a special type of button device, but because text devices are so common, we give them a separate identity.

Another button device is the *chord keyboard*. It has just five keys shaped like thin piano keys (Fig. 5.14). The device is operated by depressing several keys at once, thus "playing a chord." With five keys, 31 chords can be generated. Learning the chords takes much time, but skilled users can "play" quite rapidly, since the hand can stay in one place, resting on a table; this allows fast "touch typing" which is much harder with a standard PFK. However, chord keyboards are *not* suitable substitutes for the standard alphanumeric keyboard [KLEM71].

5.2.6 Nontraditional Devices

A number of other interaction devices are useful for graphics systems but do not easily fit into the categories we have used. For instance, speech recognizers can be used to input commands, values, etc., and are available in versions recognizing from 10 to over 1000 words. Words are defined as discrete utterances preceded and followed by 100 to 200-millisecond pauses. Continuous speech recognition, on the other hand, is mostly still in the research stage [KAPL80], although a few commercial systems have appeared.

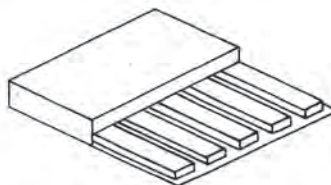


Fig. 5.14 Chord keyboard.

Most word recognizers are trained to the characteristics of an individual's speech. A word is filtered into several frequency bands. Patterns in the relative loudness of the sound of each band are matched against a dictionary of such patterns. Training the recognizer consists of establishing a dictionary of such patterns by having the user speak each word several times. Once established, a dictionary can be saved for later use. However, if a user has a cold, retraining is necessary. Speaker-independent recognizers are limited to 10 or 20 words, typically including the digits 0-9.

The converse of speech input is speech output from digitized patterns, which is eminently usable for prompts, feedback, and error messages. Synthesizers have become quite common and inexpensive and are found in several contemporary home computers. Texas Instruments' *Speak and Spell* "toy" exemplifies the capabilities becoming available. In general, we expect the use of speech input-output to increase enormously during this decade because it is natural, easy, and fast, and requires no manual dexterity.

5.3 TECHNIQUES FOR SIMULATING LOGICAL DEVICES

Many graphics programs have a need for locator, pick, valuator, keyboard and button devices, but in many graphics systems not all of these physical devices are available. It is possible, however, to provide the logical function of any of these devices with a device from any other class, although some of the simulations can be rather awkward. On the other hand, we are not advocating that separate physical devices should always be made available for each of the required logical devices. Many very easily used interactive systems have only a positioning device (usually a mouse or tablet), a keyboard, and sometimes a few special-purpose function buttons on the keyboard. These three physical devices are used to implement all the logical devices. Our purpose in the following sections is to describe some of the more common and useful input device simulations. More extensive descriptions and human factors evaluations of these techniques are found in [EMBE81, FOLE81].

Sometimes the implementations are part of the graphics package. For example, since a direct-view storage tube cannot be used with a light pen, the cross-hair cursor locator is used as a pick: the cursor is placed on top of the item to be picked. Then, when the user depresses any key, its character code and the cursor position are sent to the CPU. That position is then compared against the positions of all displayed items to find which item is closest. In other cases, the implementation will be part of the application program. One can consider the light button menu selection used in Chapter 2 to be an implementation of a button device by a pick device.

Graphics packages such as GPGS and the Core System define all five types of devices to be available and guarantee that a reasonable implementation will be provided by the package if the corresponding physical device is not available. "Reasonable" here means (a) approximately the best that can be done with the given input tools, but (b) typically less user-friendly and convenient, i.e., more time-consuming,

and (c) logically identical, so that the application program is unaware of the physical realization. This promotes program portability, but does remove control of some of the detailed, low-level interaction techniques from the application programmer. Also, certain implementations may require considerable CPU intervention, for example when simulating interrupting logical devices by using polling loops. Nevertheless, it is very cost-effective to be able to debug an application on an inexpensive storage tube, one of many in an installation, prior to running the program in production mode on one of the few more expensive high-performance displays.

5.3.1 Locator Device Simulations

Light pen. The light pen is frequently used as a locator on a vector display. A tracking cross is initially displayed somewhere on the CRT screen (Fig. 5.15). To indicate a position, the light pen is first pointed at the cross and is then used to "move" it to the desired position. This is made possible by a feedback loop which continually (each refresh cycle) moves the cross so that it is always under the light pen. The light pen's new position, at which the tracking cross is displayed, is calculated by determining which dots on the previous tracking cross were just within the pen's field of view, as in Fig. 5.16. The new center position is found by averaging the x and y coordinates of each point detected by the pen.

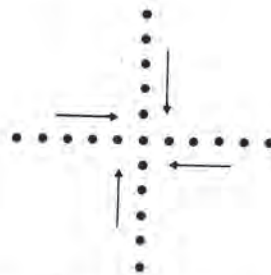


Fig. 5.15 Light-pen tracking cross: arrows show order of display. Typical height and width are one inch.

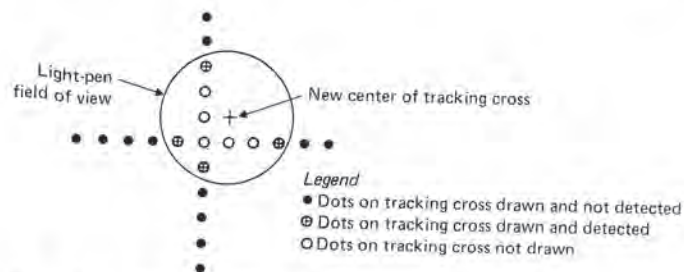


Fig. 5.16 Example of tracking.

If no points on the cross are detected, tracking has been *lost*. This can happen accidentally (if the user moves the pen faster than the tracking algorithm can move the cross) or intentionally (if the user moves the pen away from the CRT). In the latter case, the user may be indicating that the tracking cross is at its final position and that it is desired to proceed to the next interaction step.

When tracking is lost, the display system can automatically attempt to reestablish tracking to determine whether the tracking loss was accidental or intentional, that is, to try to find where the light pen is now. If the pen really has been moved away from the screen, it will not be found, and the system knows that the loss of tracking was intentional. One way to reestablish tracking is to draw a square spiral of points around the last known pen position, until the entire screen has been covered or the pen has been found, as is shown in Fig. 5.17. Another technique is to do what is effectively a raster scan of the entire screen, with scan-line spacing just slightly less than the diameter of the pen's field of vision. If the pen is still pointed at the screen, a scan line will be detected. If the vectors are drawn digitally, the detected coordinates will be immediately available, while with analog vector generators, a finer search along the scan line must be initiated in order to find the pen's horizontal position. A third approach is the *character blast*—filling the screen with consecutive characters (or points, or short lines), and stopping when one is seen by the pen. Notice that all three of these techniques can be used independently of pen-tracking to determine where on the screen the pen currently is.

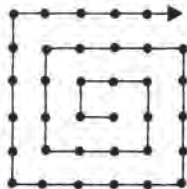


Fig. 5.17 Spiral of points used to re-establish pen tracking.

It is important that the tracking algorithm be able to move the cross at a reasonable speed, so that tracking is not easily lost. The maximum distance the cross can move per iteration of the algorithm is the length of an arm on the cross. The diagonal distance will be less. If the arms of the cross are one inch from end to end, and the cross is repositioned 30 times a second, then the maximum horizontal or vertical velocity at which the cross can be moved is 15 inches per second (one half inch per step).

One way to decrease the probability of losing tracking is to increase the speed at which the cross can move, by predicting where the pen will be during the next cycle on the basis of where it has been during the last two or three cycles. Then the cross is placed at this predicted position rather than at the position calculated by the simple averaging method described above. In the prediction it is assumed that the pen

moves with constant velocity. Thus at refresh cycle $i + 1$, the cross is placed not at the position calculated during step i , but rather at the position found by linear extrapolation from the positions of the cross at steps $i - 1$ and i , as shown in Fig. 5.18.

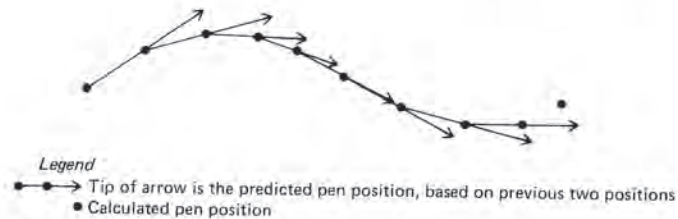


Fig. 5.18 First-order light-pen tracking prediction.

Cursor control from keyboard. A locator can also be simulated by four keys on a keyboard which control cursor movements going up, down, left, and right. Some keyboards have specially designated keys for this purpose. Holding a key down might cause rapid continuous motion of the cursor, while a quick key depression might cause only a unit move (a unit being no less than the unit of display resolution). Rapid positioning is facilitated by allowing the cursor's speed to increase as long as the key is down; when the key is released, the cursor should stop.

Position type-in. We can always simulate a locator by typing the position or orientation coordinates on a keyboard. This obviously works, but the user is forced into a very analytic approach in which desired visual results must be translated into numbers. Also, the immediate feedback of smooth dynamic position or rotation is replaced by long interaction cycles, each ending with a static display presentation. With a storage display lacking local refresh, this is the best one can expect. Otherwise, this method is very painful unless the coordinates are already precisely specified.

5.3.2 Pick Device Simulations

Cursor pick. Picks are most commonly simulated with a physical locator—tablet, mouse, etc.—with which the user moves the screen cursor to the object of interest and then pushes a button. The graphics package then searches through the (pseudo) display file for the segment closest to the cursor position. This technique is especially attractive if one of the several buttons on the mouse or tablet cursor is used to trigger the search: each button can have a different meaning, such as select, delete, move, rotate, etc. Of course too many buttons can become confusing, in which case the locator can also be used for menu selection.

5.3

The search process can be done in several ways, one of which was discussed in Chapter 4. Another is to extend the graphics package correlation table to include the *screen extent* of each segment. This screen extent is just an upright rectangle surrounding the image of a segment, and thus can be defined by $(x_{\min}, x_{\max}, y_{\min}, y_{\max})$. In the rest of this section we refer to it simply as an extent. Use of the correlation table is simple: given the simulated pick device position (x_p, y_p) , search the table for all segments such that:

$$x_{\min} < x_p < x_{\max} \quad \text{and} \quad y_{\min} < y_p < y_{\max}.$$

If only one segment satisfies the conditions, we are done. Otherwise, further processing is needed. Figure 5.19 shows why several segments might satisfy the test conditions. There are of course many cases when segments will not overlap, such as with light button symbols in a menu.

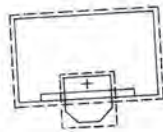


Fig. 5.19 Overlapping extents for the chair and desk symbols.

If the extents of several segments do overlap, what then? The next step is to consider the individual output primitives that make up each of the segments whose extents overlap. This will generally be a small fraction of all segments. The notion of extents can again be used, this time for individual primitives. Consider the point at (x, y) in Fig. 5.20. The extent is defined by $x_{\min} = x_p - d$, $x_{\max} = x_p + d$, $y_{\min} = y_p - d$, and $y_{\max} = y_p + d$. This extent could also be stored in a correlation table or could be computed each time it is needed by reinterpreting the DPU program. The *tolerance factor* d is chosen to allow for some reasonable amount of inaccuracy in positioning the cursor. The greater d , the less precise the user need be, but the greater the possibility that several points might be within range of the simulated pick device.

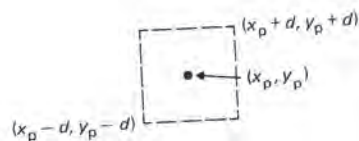


Fig. 5.20 Point extent.

For the line from $(x1, y1)$ to $(x2, y2)$ shown in Fig. 5.21, the extent is defined by

$$\begin{aligned} x_{\min} &= \min(x1, x2) - d, & x_{\max} &= \max(x1, x2) + d, \\ y_{\min} &= \min(y1, y2) - d, & y_{\max} &= \max(y1, y2) + d. \end{aligned}$$

If the application program needs to know whether an endpoint of the line or the line itself is being picked, it is simple enough first to compare the simulated pick device's position to the extents surrounding the points $(x1, y1)$ and $(x2, y2)$. Only if the endpoint comparison fails is the line comparison made.

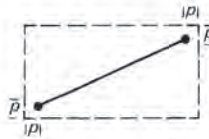


Fig. 5.21 Line extent.

This second screening, which compares the extent of each primitive to (x_p, y_p) , can also fail if there are overlapping primitive extents (Fig. 5.22). When this is the case, the final step is to determine more precisely which of the primitives is closest to the simulated pick device. For a point (x, y) this is simple:

$$D^2 = (x_p - x)^2 + (y_p - y)^2. \quad (5.4)$$

Since only comparisons of relative distances are of interest (to find which of the selected points is actually closest to (x, y)), taking the square root of this equation is unnecessary: squared distances can be compared to find a minimum just as well as the distances themselves.

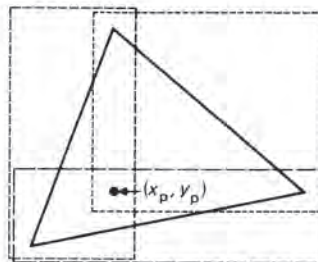


Fig. 5.22 Overlapping extents for three lines, each in separate segments.

Calculating the shortest distance from a line to a point P takes several multiplications and divisions. It is equivalent to first finding the point P' on the line which is nearest P , and then finding the distance between P and P' . One approximation is to use the minimum of the horizontal and vertical distances from the locator position to the line; this is easier to compute, but can be incorrect by $\sqrt{2}$. Forming the sum of squares to calculate D^2 can itself be time-consuming, particularly if done on a computer which lacks multiplication hardware. An excellent approximation [PIZE75] to D (not D^2) is given by:

$$\frac{|x_p - x| + |y_p - y| + 2 \cdot \max(|x_p - x|, |y_p - y|)}{3} \quad (5.5)$$

Its average error is only about 3.5%, and its maximum error is only slightly greater than 5%—quite acceptable for the approximation's intended use. Because only comparisons between distances will be made, the division by 3 need not be performed. Thus the calculation of this approximation can be faster than that of D^2 .

Whatever method is used, we compute distances from each output primitive to the simulated pick position. The first primitive found to be within d of the position is reported as having been picked. If no such primitive is found, no pick is reported.

This three-step screening process avoids detailed examination of most segments and output primitives which are far from the simulated pick's position by first examining segment extents, then primitive extents, and finally actual distances. Another approach, simpler to implement but slower in execution, examines each output primitive. The basic idea is to clip each output primitive against a small *pick window*, measuring $2d$ on a side. The first output primitive which passes through the pick window is "detected." DPUs with hardware windowing can use this approach to great advantage. As discussed in Chapter 9, extents can also be used to speed up the clipping process, by determining whether the extent and pick window overlap. Garrett reviews several of these methods in detail [GARR80a].

Many physical devices can be used to move a screen cursor to a desired target either in order to simulate a pick or as a locator. Which one is best? In a 1978 experiment, Card, English, and Burr found that a mouse is faster than an isometric joystick or cursor control keys in selecting characters and character groups from a full-page display [CARD78]. They also showed that the time to move the cursor corresponds to *Fitts' law*, which predicts that the time is proportional to the log of the ratio of the distance moved to the size of the target. The (not surprising) implication of this finding is that pick targets, be they light buttons or symbols, should be made as large as possible.

In a 1967 experiment by English, Engelbart, and Berman, the mouse was also found superior to a light pen and joystick [ENGL67]. In 1975 Goodwin found the lightpen superior to cursor control keys on a keyboard [GOOD75]. Mehr and Mehr found the trackball superior to joysticks with and without spring return and to iso-

metric joysticks [MEHR72]. While one must be careful in generalizing experimental results, the mouse certainly ranks high and the joystick low, over the range of devices studied.

Name type-in. Yet another way to simulate a pick is to assign and display names with each segment or other logical grouping of output primitives. The user then types in the name of interest. This technique is found in several commercial drafting systems; however, it is slower than the cursor pick simulation and is susceptible to typing errors. Ease of use is enhanced if the names are assigned by the user, so that they are natural to the application rather than contrived.

Successive intensification. Three buttons can simulate a pick. In this technique, each pickable entity is successively increased in intensity for a short period. When the entity to be picked brightens, the user activates one button. However, in the brief moment between the entity's brightening and the button activation, several subsequent entities are likely to have brightened, so the second button is depressed to reverse the brightening sequence, one step at a time. When the correct entity is again brightened, the third button is depressed to make the pick. This simulation has been applied successfully to pick a single molecular bond from several thousand on the screen—a case where the proximity of individual bonds precludes use of any other method. It is also appropriate with stereoscopic three-dimensional displays, as discussed in Chapter 14 [ORTO71].

5.3.3 Valuator Simulations

In graphics applications, the user must often enter numeric values which are unrelated to world coordinates. A resistor's value, Young's modulus for a beam, and a shock absorber's damping factor are all examples.

Locator axis. A single axis of a joystick, tablet, or other locator device can simulate a low-resolution valuator. Thornton has shown how to use a tablet to simulate a valuator of essentially infinite range [THOR79].

Value type-in. A value can be directly entered from a keyboard, possibly in scientific notation for very large or very small values.

Dial interaction. One of the variety of dials and scales illustrated in Fig. 5.23 can be displayed. A pick, perhaps simulated with a locator, is then used to manipulate the dial in various ways, such as dragging the pointer to a new position or picking a new tickmark on a scale. The utility of any of these valuators is often enhanced by echoing in numeric form whatever value is currently indicated. All such gauges or dials should be laid out in a natural way: we normally associate left-to-right, bottom-to-top, or clockwise movement with increasing values, and the opposite directions with decreasing values.

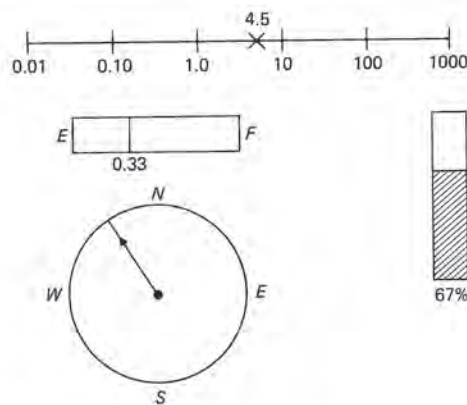


Fig. 5.23 Dials and scales.

5.3.4 Keyboard Simulations

Pattern recognition. Character recognizers have been used with interactive graphics since the early 1960s [BROW64, TEIT64]. Program and data typically occupy less than four Kbytes: several commercial microprocessor-based implementations exist. A simplified adaptation of *Teitelman's recognizer* [TEIT64], developed by Ledeen, is described in [NEWM79]. Recognition is done by comparing characteristic features (slopes, points of inflection, quadrants crossed, etc.) of characters drawn with a locator against a stored dictionary of each character's features. The recognizers can be trained to recognize different styles of block printing: the parameters of each feature are calculated from sample characters drawn by the user.

The rate at which we print is relatively slow compared to typing or writing in script—it is difficult to block print more than two or three characters a second (try it!), so recognition would not be appropriate for massive input of text. We can write script faster than we can print, but there are as yet no simple algorithms to recognize continuous script.

Keyboard menu. A menu of characters can be displayed and selected with a pick device. This technique is slow and can be considered only for very short character sequences or for operators who are known to be poor typists. The menu need not be on the screen: a keyboard can be printed on a tablet, and the tablet stylus can then be used to point at sequences of characters. This latter technique can also be used for commands, given that they change infrequently.

5.3.5 Button Simulations

Pattern recognizers. A powerful application of a locator is to recognize certain movement patterns as button activations, as illustrated in Fig. 5.24 for a queueing analysis system [IRAN71]: the eight different motions are easily recognized by a

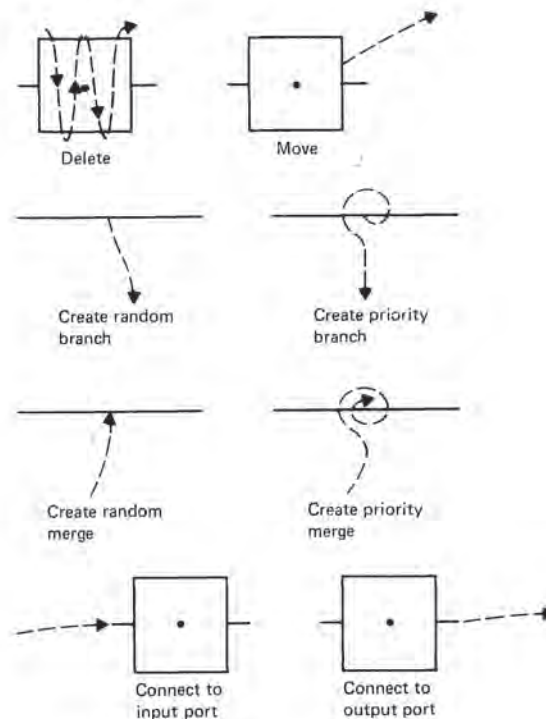


Fig. 5.24 Motions recognized as commands.

simple algorithm. Some of the meanings in this example are specialized to queueing networks, while others are quite general. A design drafting system made by Applicon recognizes several dozen such motions. The technique is especially attractive because the motion is normally performed on top of the object to be operated upon or at the position where something is to be placed, so that several parts of a complete command are input to the system at once. There are commercially available *intelligent tablets* with built-in microprocessors to recognize patterns and characters.

Menus. The menu, used in Chapter 2, is a common button simulation. The command menu contains character strings, while the furniture menu uses *icons*, or symbols, representing the various pieces of furniture. Command menus can use icons too: a straight arrow can indicate translation, a curved arrow can indicate rotation, a red paintbrush can mean MAKE IT RED.

Menus are a very important interaction technique, because memorization of command names or options is replaced by recognition. Icons are appropriate if they can be recognized more quickly than equivalent text. For example, the Xerox Star office automation workstation uses symbolic representations of in and out baskets, printers, file folders, and documents. To print a report, the user simply selects the icon for the report and moves it on top of the printer icon [SEYB81].

Menus can contain many selections. In the limit, the entire screen may be taken to show the menu (Fig. 5.25). This has devastating effects on visual continuity and maintenance of a sense of "place" in the interaction dialogue, because the visual context is lost each time the menu is shown. An abbreviated, terser menu (Fig. 5.26) is generally preferable, although the entries can be harder to understand. A *static* menu is always displayed in the same position on the screen and may or may not appear at all times. A *dynamic* (pop-up) menu appears on the screen at different locations, typically wherever the screen cursor (controlled by a tablet or other locator) happens to be. The presumption is that the operator's eyes will be on the screen cursor, so the dynamic menu is displayed there to minimize hand and eye movements.

Dynamic menus are easily implemented with raster displays: the menu replaces a rectangular part of the displayed image. With a vector display, the menu may be on top of the image, causing confusion. There are conflicting reports on the advantages of static versus dynamic menus: static menus allow use of "muscle memory" to position the pen or tablet and don't cause drastic changes to the display, while dynamic menus usually are displayed where the operator is looking, and thus visual search is minimized.



SET ENDPOINTS
SET INPUT VALUES
ORIENTATION-WITHOUT FORCE
ORIENTATION-WITH FORCE
DIRECT LINEAR FORCE
INVERSE LINEAR FORCE
DIRECT SQUARE FORCE
INVERSE SQUARE FORCE
DIRECT CUBE FORCE
INVERSE CUBE FORCE
VELOCITY DEPENDENT FORCE
PERPENDICULAR VELOCITY DEPENDENT FORCE
ARBITRARY CHARGE DISTRIBUTION
CAPACITOR
DIODE
TRIODE
NUCLEAR FORCE
TWO PLANETS
PLANE CAPACITOR
CYLINDRICAL CAPACITOR
SPHERICAL CAPACITOR
METER-FINE
METER-COARSE
SET INPUT VALUES
CHANGE PROPORTIONALITY CONSTANT
DOUBLE PROPORTIONALITY CONSTANT
HALVE PROPORTIONALITY CONSTANT
REPULSIVE FORCE
ATTRACTIVE FORCE
FIND POSITION OF GROPE DEVICE
TERMINATE

Fig. 5.25 Full-screen menu (Foley and Wallace, Proc.IEEE, Vol. 62, No. 4, 1974).

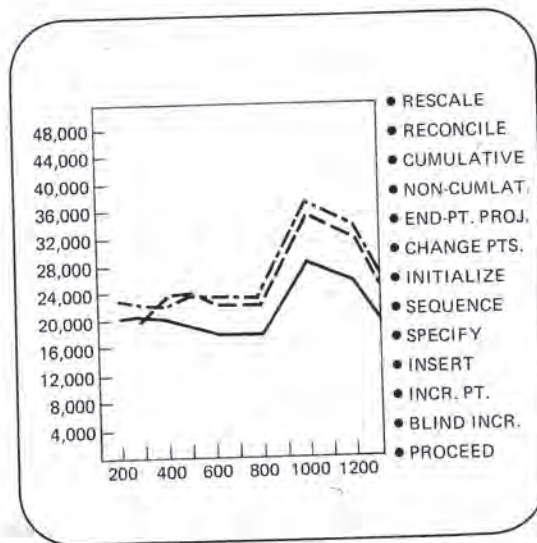


Fig. 5.26 Abbreviated menu (Foley and Wallace, Proc. IEEE, Vol. 62, No. 4, 1974).

Command type-in. Command names can be typed on a keyboard. A shorthand, useful both for poor typists and to minimize input, is used to display the commands with numbers: only the number is typed. Abbreviations can be used: they should be easy to remember, and preferably be of uniform length. The simplest rule for forming abbreviations is "fixed length," with "shortest unambiguous" abbreviations also being accepted.

5.4 INTERACTION TECHNIQUES

Interaction techniques are higher-level functions implemented through the basic devices and device simulations we have already discussed. These techniques, which are general, application-independent ways of interacting with a computer, are routinely used in many graphics application programs. The devices, device simulations, and interaction techniques are the basic building blocks from which complete interactive dialogues are designed.

5.4.1 Construction Techniques

We think of picture construction as the process of creating (or modifying) a model of an object whose image we are viewing. Common operations in this process are defining points and lines, and moving, rotating, or scaling objects. We manipulate the object by indirectly manipulating its physical appearance on the screen. *Drag-*

ging, a technique of dynamically moving an object around with a locator, was used in Chapter 2 to position furniture symbols. The locator is read frequently, preferably each refresh cycle, and its position is used to reposition the displayed object. With a vector display, the repositioning is effected relatively easily by manipulating the DPU program. The task is not quite so easy with a raster display, but can be done in a way discussed in Chapter 12. Dragging an image stored on a direct-view storage tube is impossible without using write-through mode and a refresh buffer, as found in the Tektronix 4054.

An extension of dragging is dynamic scaling and rotation of objects to a desired size and orientation. In 2D this requires that two valuator be continually read and their values used to control size and angle. Of course, if horizontal and vertical size are to be controlled simultaneously, or if a model of a 3D object is being developed, more valuator will be needed. The application of *image transformations* to clipped primitives is discussed in Chapter 8, while *geometric transformations* of world coordinate objects are discussed in Chapter 7.

Because the orientation of some complex 3D objects is difficult to perceive in a 2D projection, a *gnomon* may be used to show the orientation of the principal axes, as in Fig. 5.27. The gnomon helps facilitate 3D rotation tasks.

Lines are often constructed by *rubberbanding*. With the push of a button, the starting point of a line is read from a locator and a line is drawn from this starting point to wherever the screen cursor is positioned by the locator. As the cursor moves, so moves the line endpoint. Another push of a button or even better, release of the button depressed to start rubberbanding, and the locator is no longer coupled to the endpoint. Tablets which use a stylus with a tip switch work especially well here, as the starting and ending button pushes can come from the stylus itself. Figure 5.28 illustrates a rubberband sequence.

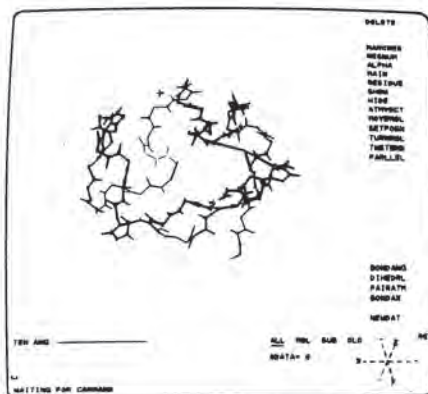


Fig. 5.27 Gnomon (lower right) helps show orientation of molecule (picture taken on Wright's GRIP system [WRIG72]).

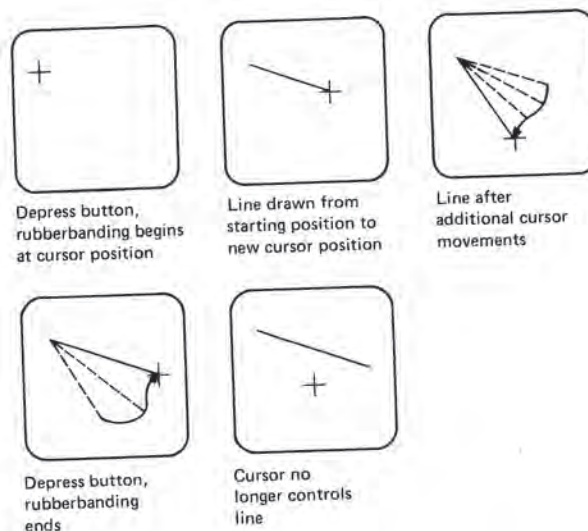


Fig. 5.28 Rubberband line drawing.

Constraints can be applied to the line resulting from rubberbanding. Figure 5.29 shows a sequence of lines drawn by using the same cursor positions as in Fig. 5.28 but with a horizontal constraint in effect. A vertical line, or a line at some other orientation, can also be drawn in this way, as can part of a circle, ellipse, or any other curve: the curve is initialized at some position, then cursor movements control how much of the curve is displayed. In general, the locator position is used as input to a constraint function which forces the appropriate feedback to be generated.

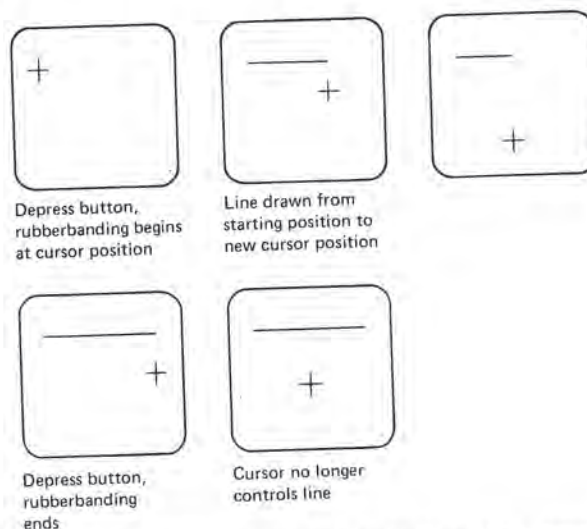


Fig. 5.29 Horizontally constrained rubberband line drawing.

Another form of constraint, applied not only in rubberbanding, is *gridding*. All endpoints of lines, etc., are constrained to fall on a grid of points (sometimes displayed as lines or dots). Gridding helps generate drawings with a neat appearance. To enforce gridding, the application program simply rounds locator coordinates to the nearest grid point. Gridding usually is applied in world coordinates. Thus the spacing of grid lines displayed on the view surface will depend on the window-to-viewport mapping. While grids are often regular and span the entire display, irregular grids, different grids in different areas, and rotated grids are all useful in creating figures and illustrations [FEIN81].

When constructing drawings, one frequently wants a new line to begin at the endpoint of or on an existing line. Positioning a cursor at such an exact point can be difficult unless gridding is in use. The difficulty is avoided by placing a programmed *gravity field* around each existing line, so that the cursor is attracted to the line as soon as it enters the gravity field. Figure 5.30 shows a line with a gravity field. The field is larger around the line's endpoints, so that matching of endpoints is especially easy. As with light-pen picking, a cluttered display makes the use of this technique difficult. A gravity field can be implemented through an approach similar to that described for simulating a pick with a locator. The tolerance factor is simply made larger for endpoints than for lines.

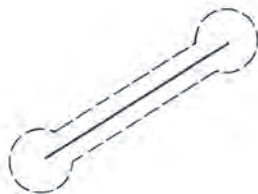


Fig. 5.30 Line surrounded by gravity field.

It is sometimes necessary to trace or draw freehand a curve or contour. *Sketching* is a technique in which a trail of lines is "sketched" along the path of a cursor, as though one were drawing a line with a pencil. In *discrete* sketching, the user pushes a button each time one line is to end and the next is to start, as in Fig. 5.31. This allows specific points to be connected by straight lines. If smooth curves are desired, then

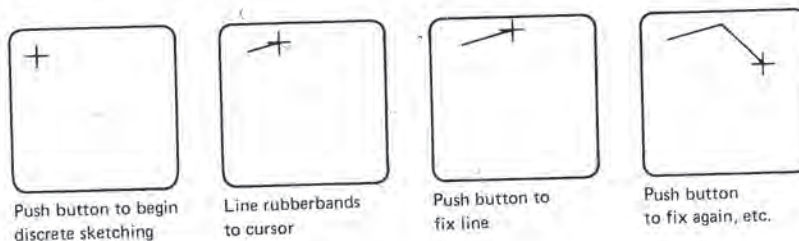


Fig. 5.31 Discrete sketching.

continuous sketching is used. With this technique, no explicit user action is required to terminate one line and start another: a series of short lines approximating a smooth curve is displayed along the path of the cursor, as in Fig. 5.32. More short lines are used where the radius of curvature is small, to give a smoother appearance.

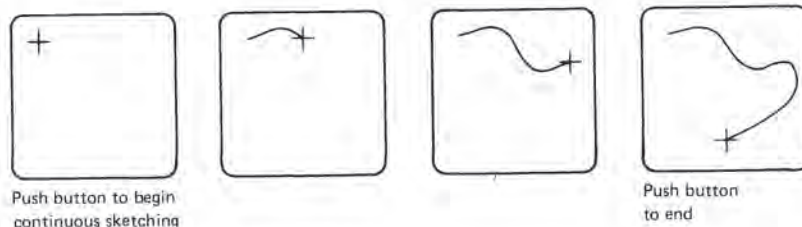


Fig. 5.32 Continuous sketching.

Dragging dynamically moves a selected symbol from one position to another, under control of a locator, as in Fig. 5.33. A button is typically pushed to start the dragging (in some cases it may be possible to have the button push also do the selecting), and then either another button push or a release of the button freezes the symbol in place, so that further movements of the locator have no effect on the symbol.

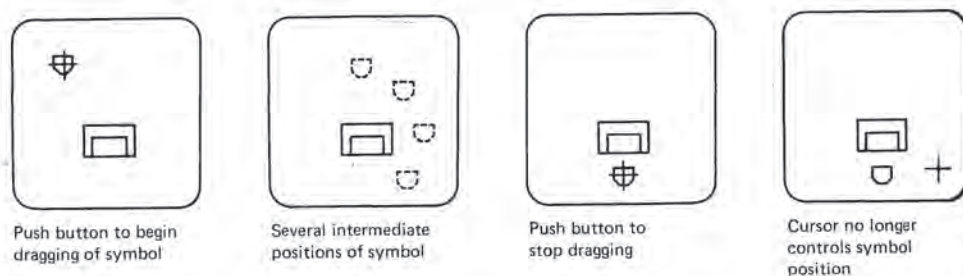


Fig. 5.33 Dragging a symbol into position.

All of these construction techniques make heavy use of immediate dynamic feedback, which can be provided only if the computer can give rapid and regular service or if the DPU has sufficient computation capability to provide the feedback on its own (see Chapter 10). This can easily be done with a small personal computer, but is often difficult with a timeshared computer. In the absence of dynamic feedback, slower and more cumbersome interaction techniques are used. For instance, rubberbanding is replaced by the capability to specify the line's endpoints, see the line drawn, and then respecify one or both of the endpoints if the line is unsatisfac-

tory. Respecifying can continue until the results are satisfactory. In essence, the dynamic feedback of rubberbanding is replaced by a much slower and less satisfactory static feedback. Consequently, the user either takes longer to complete construction tasks or produces less satisfactory results.

5.4.2 Command Techniques

We saw, in Chapter 2, the use of menus and programmed function keyboards for command entry. Many real-world applications have dozens and even hundreds of different commands—many more than can be displayed in a menu or assigned to the 16 or 32 buttons on a PFK. This plethora of commands can be dealt with by grouping them into different *phases*, or *modes*. Each phase contains a group of logically related commands, or *subphases*, which are further groupings of commands.

If a PFK is used for command entry, then the commands for each phase can be associated with a separate overlay. Designing phases which are made up of logically related commands should help minimize the frequency with which overlays need to be changed. If the application has a small nucleus of often-used commands, they should be made available in each phase and should be invoked by the same key on each overlay. Naturally, the name of the current phase or subphase should be prominently displayed, so that the operator can easily remember the context in which commands will be performed. The method of changing from one mode to another needs to be simple and obvious.

If a menu is used for command entry, then the “main” or highest-level menu will be used to select a phase. Then a second menu, with all commands in the selected phase, is shown. Several levels of phases and subphases can be used if needed: in some application programs there are as many as four or five levels in the *command hierarchy* (tree) defined by the phases. This means that four or five successive menu selections must be made in order to actually specify a command. Hierarchical menu selection can be slow and tedious for experienced users if successive menus are not displayed nearly instantaneously. One possible alternative is to allow any command name to be typed at the keyboard, whether or not it is in the currently displayed menu: now new users can pick from a menu while experienced ones can use the keyboard-oriented method. Powerful personal computers can display menus quickly, so the keyboard alternative is less attractive unless the user's hands are already on the keyboard.

5.4.3 Picking Techniques

An important part of many graphics interaction sequences is selecting the displayed object to be operated upon. Often, picks are used with a hierarchical object structure, allowing the user to pick a basic object (like a house window), a collection of basic objects (a house made up of several windows, walls, and a roof), or perhaps a collection of collections (several houses).

A pick device has no inherent notion of hierarchy, yet the user's intent must somehow be made known to the application program. One way to achieve this goal is to design the interaction commands so that the level of object to be operated on can be inferred without explicit operator action. For example, in a phase having to do with placing houses in a subdivision, the entire house would be designated, while in a phase for designing individual houses, the house components would be selected.

If the intent is not implicit, then the command language must provide the user with explicit means for conveying his purpose. Commands like MOVE WINDOW and MOVE HOUSE illustrate such means. On the other hand, if the hierarchical level at which the user operates changes relatively infrequently, it is more convenient to have a separate command which sets the level at which all ensuing picks will be made. The user respecifies the level whenever necessary.

Another approach is required if the number of hierarchical levels of objects is unknown to the system designer and is potentially large (as in a drafting system where templates can be defined to contain both other templates and basic objects like lines, points, and arcs). Two user commands are required: TRAVEL UP THE HIERARCHY and COME BACK DOWN. When the user picks something, the system highlights the lowest-level object seen. If this is what the user wants to pick, he can proceed. If not, he issues the first of the two commands: TRAVEL UP THE HIERARCHY. The entire first-level object (the house, say) of which the detected object is a part is highlighted. If this is not what the user wants, he travels up again and still more of the picture is highlighted. If he should accidentally travel too far up the hierarchy, he reverses direction with the COME BACK DOWN command.

In some text editors, there is a hierarchy of character-word-sentence-paragraph. Using the Xerox Star text editor, one selects a character by positioning the screen cursor on the character and pushing the "select" button (one of two) on the mouse. To choose the word, the user simply pushes "select" again. Further moves up the hierarchy are accomplished by further pushes.

EXERCISES

- 5.1 Solve Eqs. (5.1) for x , y , and z .
- 5.2 Solve three equations from (5.2) for x , y , and z .
- 5.3 Solve three equations from (5.3) for x , y , and z .
- 5.4 Develop an algorithm that uses all four of the Eqs. (5.2) to calculate the "best guess" position (x, y, z) of the light source. Start by solving all four combinations of three equations. This will give four different solutions for (x, y, z) . If all four are "close," an average is a good estimate. What if the solutions differ considerably?
- 5.5 Can a light pen be used with a direct-view storage tube? Why or why not?
- 5.6 How can a light pen be used as a pick device with a raster display?
- 5.7 Write a pen-tracking algorithm to do (a) tracking with spiral search if tracking is lost or (b) first- and second-order prediction.

- 8 Implement a simple feature-comparison character recognizer.
- 9 Simulate a light-pen pick with the locator and experiment with x - y correlation techniques.
- 10 Write a procedure to find the extent of a series of lines and points, allowing a tolerance of d around the outside.
- 11 Show that if we approximate distances from a point to a line by the minimum of the horizontal and vertical distances, the worst-case approximation is wrong by a factor of $\sqrt{2}$.
- 12 Describe at a high level techniques for simulating any logical device with a device from any other logical device class. Repeat this exercise, assuming the availability of a single physical device (keyboard, light pen, tablet). Devise reasonable protocols for the user to tell the system how the physical device is currently being used—as a pick, locator, etc. (This is necessary because some graphics packages allow the user to perform any one of several different actions constituting a complete command specification in any sequence: If a pick and button are both required, either may be done first.)
- 13 Write a continuous sketching subroutine which reads a locator and displays short line segments to approximate the path followed by the locator. A parameter to the subroutine is θ , the smallest angle which two successive line segments are permitted to make. That is, a sufficient number of short lines is used so that the constraint on angles is met without using more than are needed. Think of the subroutine as a filter: many points are read, and in general only some of the points become endpoints of lines.
- 14 Review and critique several experiments with interaction devices [ENGL67, GOOD75, CARD78]. Conduct a similar experiment with devices available to you.