Pick, Click, Flick!

The Story of Interaction Techniques

Brad A. Myers

ASSOCIATION FOR COMPUTING MACHINERY

Advance Praise for *Pick, Click, Flick! The Story of Interaction Techniques*

"A remarkable accomplishment and contribution from a thoughtful participant in the design revolution that made computing technology such a global success."

"Integrative insights about how interaction techniques emerged, with detailed analyses that will be a valuable resource to historians, designers, inventors, lawyers, journalists, and students."

"Brad Myers tracks the user interface failures, successes, controversies, and breakthroughs that brought the computing revolution to broad audiences. He profiles the luminaries and shows the evolving designs for widely used interaction techniques."

"Brad Myers's life-long commitment to interaction techniques makes his book a magnum opus, which will be an invaluable resource for the growing number of scholars, designers, and students eager to learn about the history of user interface design."

- Ben Shneiderman, Retired Emeritus Professor, Computer Science, University of Maryland, and former Founding Director: Human Computer Interaction Lab

"*Pick, Click, Flick!* is an impressive reference manual of the many years of interaction design development. Look up any subject and you will learn the history, the principles, and be provided with numerous examples of actual implementation and usage. This is not a book to sit down and read. It is far more important than that. It is a reference book, invaluable when questions arise, whether while you are busy designing something, or learning, or teaching, where assigning sections of the reference will be a valuable resource and learning tool for students. Few will read the entire book, but everyone doing or interested in interaction design should

have a copy on their shelves (or computer files), ready for consultation whenever questions arise. Brad Myers has provided a great service to the interaction community."

personal note:

"Every time I open your 700-page masterpiece, I see that it covers everything, from the early days to now. That was a daunting task to accomplish, but it is also a daunting task for the reviewer who is obligated to read it all, despite my warning that 'this is not a book to sit down and read,' but rather as a resource to consult. Frequently."

- Don Norman, Author of *Design for a Better World*, Distinguished Prof. Emeritus, Design Lab, University of California, San Diego

"Every UX professional should immerse themselves in this book. Not only does it unravel the fascinating and complex history of GUI widgets that will captivate any user interface nerd, but it also stands as the definitive guide to an incredibly diverse array of interaction techniques. This is not just an engaging read; it's an essential toolkit. By delving into these intricate details, you're not merely learning—you're evolving into a more refined and effective designer."

- Jakob Nielsen, Principal, Nielsen Norman Group

A few years ago, as the field of Human–Computer Interaction was emerging with its first tentative results, we worried that the field wouldn't be able to produce sufficient findings to achieve traction. The problem now is just the opposite—the volume and heterogeneity of available results overwhelms the designer and the engineer, making it difficult for them to access what is known about input devices and techniques. Professor Myers' book takes us on a romp from the most important devices of the recent past (like the Xerox mouse) to tactile keyboards to mixed reality (like Microsoft's Mixed Reality Toolkit). Professor Myers is just the guy you want to have with you on your journey of finding the right device for your project. He seems to have deep experience with everything. We are entering a new era of Human–Computer Interaction. I can think of no better guide to this era than *Pick, Click, Flick!*"

- Stuart Card, Adjunct Professor, Computer Science Dept., Stanford University; and Retired Senior Research Fellow at Xerox PARC

"No one has done more than Brad Myers to turn the art of computer user interface design into a science. Now with this masterful book, he has put a fitting cap on his eminent career. He covers all of the important developments in interface design in the last fifty years and describes them clearly and objectively. He also includes many of the most important innovators, many of whom are personal friends. It is a remarkably thorough coverage of the history of a vibrant inventive field."

- David Canfield Smith, Retired; former leader of the User Interface Design Team for the Xerox Corporation's Star

(Smith's interview is Chapter 24)

"'Interaction Techniques' are pivotal in defining the contours of our digital experiences. For over 50 years, Professor Brad Myers, a distinguished luminary in the realm of Human–Computer Interaction (HCI), has been at the forefront, pushing boundaries and making notable advancements in the study of Interaction Techniques. This book offers a thorough chronicle of the evolution of interaction techniques research during the era of ubiquitous computing. It's a must-read for HCI students, researchers, and practitioners dedicated to sculpting the future landscape of human–computer interaction."

- Hiroshi Ishii, Jerome B. Wiesner Professor of Media Arts and Sciences; Associate Director of MIT Media Laboratory; Director of Tangible Media Group; Massachusetts Institute of Technology

Pick, Click, Flick!

ACM Books

Editors in Chief

Sanjiva Prasad, *Indian Institute of Technology (IIT) Delhi, India*
Marta Kwiatkowska, *University of Oxford, UK*

ACM Books is a new series of high-quality books for the computer science community, published by ACM. ACM Books publications are widely distributed in both print and digital formats through booksellers and to libraries (and library consortia) and individual ACM members via the ACM Digital Library platform.

Digital Dreams Have Become Nightmares: What We Must Do
Ronald M. Baecker, *University of Toronto (Computer Science Department)*
with Jonathan Grudin, *University of Washington (Information School)*
2024

Spatial Gems, Volume 2
Editors: John Krumm, *Viterbi School of Engineering, University of Southern California, Los Angeles, CA, USA*
Andreas Züfle, *Emory University, Atlanta, GA, USA*
Cyrus Shahabi, *Computer Science Department, University of Southern California, Los Angeles, CA, USA*
2024

From Algorithms to Thinking Machines: The New Digital Power
Domenico Talia, *University of Calabria*
2023

The Societal Impacts of Algorithmic Decision-Making
Manish Raghavan, *Massachusetts Institute of Technology, Sloan School of Management and Department of Electrical Engineering and Computer Science*
2023

Geospatial Data Science: A Hands-on Approach for Building Geospatial Applications Using Linked Data Technologies
Editor: Manolis Koubarakis, *National and Kapodistrian University of Athens, Greece*
2023

Linking the World's Information: Essays on Tim Berners-Lee's Invention of the World Wide Web
Editors: Oshani Seneviratne, *Rensselaer Polytechnic Institute*
James Hendler, *Rensselaer Polytechnic Institute*
2023

Logic, Automata, and Computational Complexity: The Works of Stephen A. Cook
Editor: Bruce M. Kapron, *University of Victoria*
2023

Pick, Click, Flick!

The Story of Interaction Techniques

Brad A. Myers
Carnegie Mellon University

ACM Books #57

Copyright © 2024 by Association for Computing Machinery

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, photocopy, recording, or any other except for brief quotations in printed reviews—without the prior permission of the publisher.

Designations used by companies to distinguish their products are often claimed as trademarks or registered trademarks. In all instances in which the Association of Computing Machinery is aware of a claim, the product names appear in initial capital or all capital letters. Readers, however, should contact the appropriate companies for more complete information regarding trademarks and registration.

Pick, Click, Flick! The Story of Interaction Techniques
Brad A. Myers

books.acm.org
http://books.acm.org

ISBN: 979-8-4007-0949-4 hardcover
ISBN: 979-8-4007-0947-0 paperback
ISBN: 979-8-4007-0948-7 EPUB
ISBN: 979-8-4007-0950-0 eBook

Series ISSN: 2374-6769 print 2374-6777 electronic

DOIs:

10.1145/3617448 Book
10.1145/3617448.3617449 List of Figures
10.1145/3617448.3617450 List of Tables
10.1145/3617448.3617451 Acknowledgments
10.1145/3617448.3617452 Preface
10.1145/3617448.3617453 Chapter 1
10.1145/3617448.3617454 Chapter 2
10.1145/3617448.3617455 Chapter 3
10.1145/3617448.3617456 Chapter 4
10.1145/3617448.3617457 Chapter 5
10.1145/3617448.3617458 Chapter 6
10.1145/3617448.3617459 Chapter 7
10.1145/3617448.3617460 Chapter 8
10.1145/3617448.3617461 Chapter 9
10.1145/3617448.3617462 Chapter 10
10.1145/3617448.3617463 Chapter 11
10.1145/3617448.3617464 Chapter 12
10.1145/3617448.3617465 Chapter 13
10.1145/3617448.3617466 Chapter 14
10.1145/3617448.3617467 Chapter 15
10.1145/3617448.3617468 Chapter 16
10.1145/3617448.3617469 Chapter 17
10.1145/3617448.3617470 Chapter 18
10.1145/3617448.3617471 Chapter 19
10.1145/3617448.3617472 Chapter 20
10.1145/3617448.3617473 Chapter 21
10.1145/3617448.3617474 Chapter 22
10.1145/3617448.3617475 Chapter 23
10.1145/3617448.3617476 Chapter 24
10.1145/3617448.3617477 Chapter 25
10.1145/3617448.3617478 Chapter 26
10.1145/3617448.3617479 Chapter 27
10.1145/3617448.3617480 Chapter 28
10.1145/3617448.3617481 Chapter 29
10.1145/3617448.3617482 Chapter 30
10.1145/3617448.3617483 Chapter 31
10.1145/3617448.3617484 Chapter 32
10.1145/3617448.3617485 Chapter 33
10.1145/3617448.3617486 Chapter 34
10.1145/3617448.3617487 Chapter 35
10.1145/3617448.3617488 Chapter 36
10.1145/3617448.3617489 Chapter 37
10.1145/3617448.3617490 Chapter 38
10.1145/3617448.3617491 Chapter 39
10.1145/3617448.3617492 Chapter 40
10.1145/3617448.3617493 Bibliography
10.1145/3617448.3617494 Bios/Index

A publication in the ACM Books series, #57
Editors in Chief: Sanjiva Prasad, *Indian Institute of Technology (IIT) Delhi, India*
Marta Kwiatkowska, *University of Oxford, UK*
Area Editor: Michel Beaudouin-Lafon, *Université Paris-Sud*

This book was typeset in Arnhem Pro 10/14 and Flama using LuaTEX.

First Edition

10 9 8 7 6 5 4 3 2 1

To Bernita Myers

Table of Contents

List of Figures xxxi
List of Tables lv
Acknowledgments lvii
Preface lxi
Chapter 1 Introduction 1

PART I HISTORY AND BACKGROUND 19

Chapter 2 History of Desktop Devices 21
Chapter 3 History of Mobile Devices 59

PART II SPECIFIC INTERACTION TECHNIQUES 89

Chapter 4 Pointing Devices 91
Chapter 5 Scrolling and One-Dimensional Input 119
Chapter 6 Percent-done Progress Indicators 157
Chapter 7 Menus 163
Chapter 8 Text Entry and Editing for Computers and Handhelds 193
Chapter 9 Forms, Property Sheets, and Dialog Boxes 249
Chapter 10 Creating, Selecting, and Manipulating Objects 261
Chapter 11 Desktop Metaphors, Window Managers, and Icons 293
Chapter 12 Interactions in 3D, Virtual and Augmented Reality, and Games 325
Chapter 13 Ubiquitous Computing, Physical Widgets, Sensors, Wearables, and Internet of Things (IoT) 353
Chapter 14 Gesture Recognition 365
Chapter 15 Undo, Redo, Repeat, Cancel, Command History, and Macro Recording 383
Chapter 16 Interaction Techniques for Help 413
Chapter 17 Interaction Techniques for People with Disabilities 423
Chapter 18 Intelligent Interfaces: Artificial Intelligence in Interaction Techniques, Speech Recognition, and Conversational Agents 441

PART III USING, DESIGNING, AND IMPLEMENTING INTERACTION TECHNIQUES 463

Chapter 19 Formal and Informal Models, Metaphors, and Patterns for the Design of Interaction Techniques 465
Chapter 20 Tools and Models for Using and Implementing Interaction Techniques 483
Chapter 21 Methods for Designing and Evaluating Interaction Techniques 525
Chapter 22 Legal Issues and Patents 531

PART IV INTERVIEWS 541

Chapter 23 Interview: Larry Tesler 543
Chapter 24 Interview: David Canfield Smith 551
Chapter 25 Interview: Bill Buxton 559
Chapter 26 Interview: Norm Cox 563
Chapter 27 Interview: Dan Bricklin 569
Chapter 28 Interview: Bill Atkinson 575
Chapter 29 Interview: Bruce Horn 585
Chapter 30 Interview: Rob Haitani 591
Chapter 31 Interview: Tandy Trower 597
Chapter 32 Interview: Loren Brichter 609
Chapter 33 Interview: Don Hopkins 615
Chapter 34 Interview: Ted Selker 619
Chapter 35 Interview: Shumin Zhai 627
Chapter 36 Interview: Gregg Vanderheiden 631
Chapter 37 Interview: Chris Harrison 637
Chapter 38 Interview: Julia Schwarz 641

PART V FUTURE WORK AND CONCLUSIONS 649

Chapter 39 Future of Interaction Techniques 651
Chapter 40 Conclusions 657
Bibliography 659
Index 703
Author's Biography 741

Detailed Table of Contents

List of Figures xxxi

List of Tables lv

Acknowledgments lvii

Preface lxi

Overview: lxi
Who is this book for: lxii
Use in classes: lxiii

Chapter 1 **Introduction** 1

1.1 Importance of the Topic 1
1.2 What Are Interaction Techniques? 5
1.3 Formal Definitions 5
1.3.1 User and Designer 5
1.3.2 User Interface and Usability 6
1.3.3 Pointing Device 7
1.3.4 Input and Output 7
1.3.5 Graphical User Interface 7
1.3.6 Human–Computer Interaction 7
1.3.7 States and State Diagrams 8
1.3.8 Interaction Technique 8
1.3.9 Other Names 9
1.4 What Is *Not* an Interaction Technique? 10
1.5 Why Am I Writing this Book? 11
1.6 Value of Interaction Techniques 14
1.7 Need for New Interaction Techniques 14
1.8 Why Are Interaction Techniques Hard to Design? 15
1.8.1 Many Needs 15

1.8.2 Details of the Behavior 15
1.8.3 Abort, Cancel, and Undo 16
1.8.4 The Look 17
1.8.5 Localization and Internationalization 18
1.9 Organization of the Book 18

PART I HISTORY AND BACKGROUND 19

Chapter 2 History of Desktop Devices 21
2.1 Introduction 21
2.2 Computer Hardware 21
2.2.1 Mainframes 21
2.2.2 Personal Workstations 24
2.2.3 Personal Computers 25
2.2.4 Other Devices 25
2.3 Display Technologies 26
2.4 Important Systems 28
2.4.1 Sketchpad, 1963 28
2.4.2 NLS, 1968 30
2.4.3 Xerox Alto, 1970s 31
2.4.3.1 Smalltalk, 1976 31
2.4.3.2 Bravo Text Editor, 1974 32
2.4.3.3 Various Drawing and Painting Programs, 1970s 33
2.4.3.4 Xerox Programming Environments, 1980–1981 35
2.4.4 VisiCalc, 1977 35
2.4.5 Spatial Data Management System, 1978 36
2.4.6 Xerox Star, 1981 37
2.4.7 Apple Lisa (1983) and Macintosh (1984) 38
2.4.8 IBM PCs (1981) and Microsoft Windows (1985) 40
2.4.9 Apple HyperCard (1987) 40
2.4.10 Alternate Reality Kit, 1986 42
2.4.11 NeXT, 1985–1997 42
2.4.12 SunWindows, NeWS, Andrew, X, OpenLook, and Motif 44
2.4.13 World Wide Web 46
2.5 Brief Overview of Human–Computer Interaction Technology 48
2.5.1 Interaction and Application Styles 50
2.5.1.1 Command Line Interfaces 50
2.5.1.2 Form Fill-in 50
2.5.1.3 Direct Manipulation of Graphical Objects 50
2.5.1.4 Window Managers 52

2.5.1.5 Drawing and Painting Programs 52
2.5.1.6 Text (Document) Editing 53
2.5.1.7 Spreadsheets 53
2.5.1.8 Hypertext and the World Wide Web 53
2.5.1.9 Computer-aided Design 54
2.5.1.10 Video Games 54
2.5.1.11 Gesture Recognition 54
2.5.1.12 Multimedia 55
2.5.1.13 3D 55
2.5.1.14 Virtual Reality and Augmented Reality 55
2.5.1.15 Computer-supported Cooperative Work 56
2.5.1.16 Natural Language and Speech 56
2.5.2 Implementation Techniques 56
2.5.2.1 User Interface Management Systems and Toolkits 56
2.5.2.2 Interface Builders 57
2.5.2.3 Component Architectures 57
2.6 Chapter Conclusion 58

Chapter 3 History of Mobile Devices 59

3.1 Introduction 59
3.2 Definitions 60
3.2.1 Mobile Devices and Handheld Devices 60
3.2.2 Touchscreens and Touchpads 61
3.2.3 Palm-size Devices, Tablets, and Wearables 61
3.2.4 Pen Computing 62
3.2.5 Gestures 63
3.2.6 Handwriting Recognition and Hand Printing Recognition 63
3.2.7 Personal Digital Assistants 63
3.2.8 Mobile Phones, Feature Phones, and Smartphones 63
3.3 Calculators 64
3.4 Ubiquitous Computing 64
3.5 Pen Operating Systems 65
3.6 GRiDPad 67
3.7 Microsoft Pen Windows 68
3.8 Apple Newton 68
3.9 General Magic's "Magic Cap" OS 70
3.10 Early Wireless Phones 70
3.11 Early Smartphones 71
3.12 Wireless Networks 73
3.13 Palm 74

3.14 Windows CE, PocketPC, etc. 78
3.15 RIM Blackberry 80
3.16 Windows TabletPC 81
3.17 iPod 82
3.18 iPhone 83
3.19 Android 84
3.20 iPad 85
3.21 Chapter Conclusion 87

PART II SPECIFIC INTERACTION TECHNIQUES 89

Chapter 4 Pointing Devices 91
4.1 Introduction 91
4.2 Cursor and Selection 91
4.3 Mouse Buttons, Press, Hold, Click, Tap, and Drag—Definitions 93
4.4 Lightpen 94
4.5 Arrow Keys 95
4.6 Mouse 96
4.7 Joysticks and Trackballs 97
4.8 IBM TrackPoint 100
4.9 Graphics Tablets 100
4.10 Touchscreens 102
4.11 Remote Interaction 104
4.12 Dimensions for Pointing Devices 105
4.12.1 Direct versus Indirect 105
4.12.2 Absolute versus Relative 106
4.12.3 How Many States Are Supported? 106
4.13 Other Issues 109
4.14 Testing for Speed and Accuracy 110
4.14.1 Fitts' Law 111
4.15 Research About Pointing 115
4.16 Summary of the Considerations for Pointing 116
4.17 Chapter Conclusion 117

Chapter 5 Scrolling and One-Dimensional Input 119
5.1 Introduction 119
5.2 Auto-Scroll 120
5.3 Knobs 121
5.4 Physical One-Dimensional Sliders 123

5.5 Keys and Physical Buttons 125
5.6 Scrollbars 126
5.6.1 Bravo's Scrollbar, 1974 127
5.6.2 Smalltalk's Scrollbar, 1977 128
5.6.3 Interlisp-D, Symbolics Lisp Machine, XDE Scrollbars, and Andrew, 1980–1984 129
5.6.4 Xerox Star (1981) and Viewpoint (1985) 131
5.6.5 Apple Lisa (1983) and Macintosh (1984) Scrollbars 133
5.6.6 Windows, OpenLook, and NeXT Scrollbars 135
5.6.7 Mobile Device Scrollbars 137
5.6.8 iPhone Scrollbars 138
5.6.9 Variations on Scrollbars 138
5.6.9.1 Showing Other Information in the Bar 138
5.6.9.2 Auto-Hide 140
5.6.9.3 Infinite Scrolling 140
5.7 Using a "Hand" to Scroll 141
5.8 Rate-Controlled Scrolling 143
5.9 Gestures for Scrolling 144
5.9.1 Flick-to-Scroll 144
5.9.2 One- and Two-Finger Scrolling 145
5.10 Sliders 146
5.11 Research on Scrolling and Sliders 149
5.11.1 Evaluating Scrolling Speed 149
5.11.2 Research on Scrolling and Sliders 152
5.12 Summary of the Considerations for Scrolling 153
5.13 Chapter Summary 155

Chapter 6 Percent-done Progress Indicators 157

6.1 Introduction 157
6.2 History 158
6.3 Implementation 160
6.4 Variations 160
6.5 Design Considerations and Conclusions 162

Chapter 7 Menus 163

7.1 Definitions 163
7.2 Different Categories of Menus 164
7.3 Dimensions 169
7.3.1 How Many Options Can Be Selected? 169
7.3.2 How Many and What Kind of Options Are There? 169

7.3.3 Fixed or Popup? 170
7.3.4 Invalid? 171
7.3.5 Cancellable? 172
7.3.6 Submenus 172
7.3.7 Graphic Design 173
7.3.8 Default Value? 176
7.3.9 Layout 178
7.3.9.1 Pie Menus and Marking Menus 179
7.3.10 Where the Menu Is Displayed 180
7.3.11 Behaviors 181
7.3.11.1 How Many States 181
7.3.11.2 Moving from One Item to Another 182
7.3.11.3 Pointer Operation 182
7.3.11.4 Keyboard Operation 183
7.4 Examples of Historical Menus 185
7.5 Research on How to Organize Menu Contents 188
7.6 Summary of Design Considerations for Menus 190
7.7 Chapter Conclusions 191

Chapter 8 Text Entry and Editing for Computers and Handhelds 193
8.1 Introduction 193
8.2 Text Entry for Desktops 194
8.2.1 Typewriters 194
8.2.2 Chinese Text Entry 195
8.2.2.1 The Chinese Language 195
8.2.2.2 Chinese Keyboard Typing Methods 198
8.2.2.3 Chinese Text Input Interaction Techniques 199
8.2.3 Chorded Text Entry 201
8.2.4 Other Text Entry for Desktops 201
8.3 Text Entry for Mobile Devices 201
8.3.1 Hand Printing and Handwriting 202
8.3.2 Gestural Alphabets 203
8.3.3 Keypads 205
8.3.3.1 Multi-tap 206
8.3.3.2 T9 207
8.3.4 Small Physical Keyboards 208
8.3.5 Portable Chording Keyboard 208
8.3.6 Virtual Keyboards 209
8.3.6.1 Research on Virtual Keyboard Layouts 212
8.3.6.2 Gestures Over Virtual Keyboards 213

8.3.7 Text Entry on Wearables and Watches 215
8.4 Auto-Fill, Auto-Correction, and Autocomplete 217
8.4.1 Auto-Fill 218
8.4.2 Spell Checking and Auto-Correction on Regular Computers 218
8.4.3 Autocomplete on Regular Computers 220
8.4.4 Auto-Correction and Autocompletion on Mobile Devices 223
8.5 Measuring Text Entry Speed and Accuracy 227
8.5.1 Speed 227
8.5.2 Learning 228
8.5.3 Accuracy 229
8.5.4 Confounds from Auto-Correct and Autocomplete 231
8.5.5 Target Text for Testing 231
8.5.6 Summary of Speed and Accuracies of Popular Methods 232
8.6 Text Editing and Formatting 232
8.6.1 Brief History of Text Editing and Formatting 233
8.6.2 Editing Systems for Code 236
8.6.3 Text Editing Techniques 238
8.6.3.1 Moded versus Modeless Editing 238
8.6.3.2 Positioning the Text Cursor and Selecting Text on Desktops 238
8.6.3.3 Positioning the Text Cursor and Selecting Text on Mobile Devices 241
8.6.3.4 Operations for Editing 243
8.6.4 Multi-User Text Editing 245
8.7 Text Entry Widgets 245
8.8 Typeahead 246
8.9 Summary of Considerations for Designing New Text Entry Methods 247
8.10 Chapter Conclusions 248

Chapter 9 Forms, Property Sheets, and Dialog Boxes 249

9.1 Introduction 249
9.2 Xerox Star Property Sheets 249
9.3 Apple's Dialog Boxes 250
9.4 Tabbed Dialog Boxes 252
9.5 Toolbars and Ribbons 253
9.6 Typical Built-In Dialog Boxes 254
9.7 Web Pages as Forms 255
9.8 Dialog Boxes on Mobile Devices 255
9.9 Research About Dialog Boxes 255

9.10 Summary of Design Considerations for Dialog Boxes 258
9.11 Chapter Conclusions 260

Chapter 10 Creating, Selecting, and Manipulating Objects 261

10.1 Introduction 261
10.2 Syntax of Operations 261
10.3 Graphical Object Hierarchies 263
10.4 Creating New Objects and Painting 264
10.5 Selection of Graphics 264
10.5.1 Selecting One or Multiple Objects 264
10.5.2 Clicking on Objects 265
10.5.3 Selecting in a Region 266
10.5.4 Feedback for Selecting Objects 267
10.5.5 Selecting Pixels for Painting Programs 268
10.6 Specifying Position, Size, and Shape 269
10.6.1 Coordinate Systems 270
10.6.2 Global Parameters—Feedback and Aborting 270
10.6.3 Size and Position for Bounding Boxes 271
10.6.4 Size and Position for Lines, Curves, Polylines, and Freehand Scribbles 275
10.6.5 Grids 276
10.6.6 Snapping 276
10.7 Specifying Colors 278
10.8 Specifying Other Graphical Properties 279
10.9 Specifying Animations and Behaviors 280
10.10 Selection in Spreadsheets 281
10.11 Selections in Linear and Hierarchical Lists 282
10.12 Selection in Consumer Electronics 282
10.13 Brief History of Graphical Editing 284
10.14 Research in Selection and Editing Objects 288
10.15 Summary of Considerations for Selecting and Creating Objects 290
10.16 Chapter Conclusion 291

Chapter 11 Desktop Metaphors, Window Managers, and Icons 293

11.1 Introduction 293
11.2 Definitions 293
11.3 A Brief History of Window Managers 294
11.4 Input Focus 296
11.5 Overlapping Windows versus Tiled versus Tabbed versus Single Window 298
11.6 Icons 303

11.7 Task Bar and Dock 306
11.8 Decorations for Individual Windows 309
11.9 Interaction Techniques for Windows 310
11.10 Collections of Windows 313
11.11 Alternatives 315
11.11.1 Zoomable User Interface 315
11.11.1.1 SketchPad 316
11.11.1.2 Spatial Data Management System 316
11.11.1.3 PAD, PAD++, and Jazz 316
11.11.2 Timeline Systems 317
11.11.3 The Open Data Model 318
11.11.4 Microsoft Bob 320
11.11.5 Other Organizations 321
11.12 Summary of Considerations for Desktops, Windows, and Icons 321
11.13 Chapter Conclusion 323

Chapter 12 Interactions in 3D, Virtual and Augmented Reality, and Games 325

12.1 Introduction 325
12.2 Definitions 325
12.3 Why Are 3D-IxTs Difficult 328
12.4 Input Devices for 3D IxTs 332
12.4.1 1D and 2D Input Devices for 3D 332
12.4.2 Early 3D Position Sensors and HMDs 332
12.4.3 Early Tracking of the User's Fingers—Gloves 333
12.4.4 Tracking a Wand 335
12.4.5 Camera-Based Techniques 336
12.4.6 Input Techniques with Haptic Feedback 337
12.4.7 Other Sensors 339
12.5 3D Interaction Techniques 340
12.5.1 IxTs for Games 341
12.5.2 IxTs Using the Mouse 344
12.5.3 IxTs Using 3D Inputs 347
12.5.4 3D Widgets 349
12.6 Summary of Considerations for 3D Interactions 349
12.7 Chapter Conclusion 351

Chapter 13 Ubiquitous Computing, Physical Widgets, Sensors, Wearables, and Internet of Things (IoT) 353

13.1 Introduction 353
13.2 Definitions 353
13.3 Turtles to Robots 355

13.4 Physical Objects 358
13.5 Wearables 361
13.6 Handheld Projectors 361
13.7 IoT 362
13.8 Summary of Considerations 363
13.9 Chapter Conclusions 364

Chapter 14 Gesture Recognition 365

14.1 Introduction and Definitions 365
14.2 Advantages and Disadvantages of Gestures 366
14.2.1 Advantages 366
14.2.2 Disadvantages 367
14.3 Applications for Gestures 368
14.3.1 Gestures for Text Entry 369
14.3.1.1 Recognizing Sign Language Gestures 370
14.3.2 Gestures for Proofreading and Editing Text 370
14.3.3 Gestures for Object Entry and Editing 371
14.3.4 Gestures for Other Commands 372
14.4 Gesture Recognition Algorithms 373
14.5 Evaluating and Improving the Learnability and Accuracy of Gestures 378
14.6 Funny 379
14.7 Summary of Considerations for Gesture Recognition 379
14.8 Chapter Conclusions 381

Chapter 15 Undo, Redo, Repeat, Cancel, Command History, and Macro Recording 383

15.1 Introduction 383
15.2 Definitions 384
15.3 History of Undo/Redo 385
15.4 Cancel or Abort 388
15.5 Unit of Undo 390
15.6 What Can Be Undone 391
15.7 Redo/Repeat 392
15.8 Undo/Redo Models 393
15.8.1 Single-level Undo 393
15.8.2 Multi-level Undo 394
15.8.2.1 Linear Model 394
15.8.2.2 Emacs Model 395
15.8.2.3 Tree Model 397
15.8.3 Selective Undo 399
15.8.3.1 Selective Undo by Region 400

15.8.3.2 Script Model 401
15.8.3.3 Inverse Model 403
15.9 The Back Button 405
15.10 Multi-user Undo 406
15.11 Scope of Undo 408
15.12 Implementing Undo and the Command Object Pattern 408
15.13 Macro Recording and Programming by Example 410
15.14 Summary of Considerations for Undo/Redo 411
15.15 Chapter Conclusions 412

Chapter 16 Interaction Techniques for Help 413

16.1 Introduction 413
16.2 Status Bar 414
16.3 Tooltips 415
16.4 Autocomplete as Help 417
16.5 Help Modes 417
16.5.1 Ask About a Screen Item 417
16.5.2 Stencils 420
16.6 Considerations for Help and Chapter Conclusions 421

Chapter 17 Interaction Techniques for People with Disabilities 423

17.1 Introduction 423
17.2 Related Concepts and Definitions 425
17.3 Kinds and Causes of Disabilities 425
17.3.1 Muscular Problems 425
17.3.2 Visual Impairment and Blindness 427
17.3.3 Deafness and Hearing Loss 428
17.4 Interaction Techniques Related to Disabilities 429
17.4.1 Adaptive Input Devices 429
17.4.2 Selecting Items by Scanning 430
17.4.3 Text Entry 431
17.4.4 Augmentative and Alternative Communication Devices 432
17.4.5 Brain–Computer Interfaces 432
17.5 Screen Readers 433
17.6 Adaptations Built into PC and Macintosh 435
17.7 Adaptations Built into Smartphones and Tablets 435
17.8 Web, App, and Consumer Electronics Accessibility 437
17.9 Summary of Consideration for Interaction Techniques for People with Disabilities 439
17.10 Chapter Conclusions 440

Chapter 18 **Intelligent Interfaces: Artificial Intelligence in Interaction Techniques, Speech Recognition, and Conversational Agents** 441

18.1 Introduction 441
18.2 Definitions 442
18.3 Speech and Natural Language Interfaces 443
18.3.1 Kinds of Dialogs and What Can Be Said 444
18.3.2 How Users Provide Input 445
18.3.3 Feedback, Verifying, and Correcting 447
18.3.4 System Actions 449
18.4 Multi-Modal Interaction Techniques 450
18.5 Embodied Intelligent Agents 452
18.6 Algorithms and Machine Learning for Content Selection 455
18.7 Other "Intelligent" Interfaces 456
18.8 Challenges, Risks, and Controversy About Intelligent Interfaces 458
18.9 Summary of Considerations for Intelligent Interfaces 461
18.10 Chapter Conclusions 462

PART III **USING, DESIGNING, AND IMPLEMENTING INTERACTION TECHNIQUES** 463

Chapter 19 **Formal and Informal Models, Metaphors, and Patterns for the Design of Interaction Techniques** 465

19.1 Introduction 465
19.2 "Model" 466
19.3 Metaphors and UI Design Patterns for UIs 467
19.4 State Transition Diagrams 469
19.5 Input Device Models 471
19.5.1 Foley and Wallace (1974) and Chan (1984) 471
19.5.2 Buxton's Classification (1983) 472
19.5.3 Card, Mackinlay, Robertson Model (1990) 473
19.6 Pragmatic, Lexical, Syntactic, Semantic, and Conceptual Level Model 473
19.7 Human Performance Models 476
19.7.1 Keystroke-level Model 476
19.7.2 Goals, Operators, Methods, and Selection Rules 478
19.8 Other Models 480
19.9 Considerations for Models 480
19.10 Chapter Conclusions 481

Chapter 20 Tools and Models for Using and Implementing Interaction Techniques 483

20.1 Introduction 483
20.2 Why Hard to Implement 484
20.3 Definitions 485
20.4 Why Tools Are Important 488
20.5 Toolkits and Frameworks 489
20.5.1 Look and Feel 490
20.5.2 Frameworks 490
20.5.3 Toolkit and Framework Class Hierarchies 491
20.5.4 Research Toolkits and Frameworks 492
20.6 Geometry and Layout Management 492
20.6.1 Absolute Layout 492
20.6.2 General Constraints 493
20.6.3 Layout Managers 494
20.6.3.1 Special Values for Coordinates 494
20.6.3.2 Special Kinds of Containers 495
20.6.3.3 Stretchy Spacers 496
20.7 Resource Files and Internationalization 496
20.8 Interactive and Graphical Tools for Building UIs 499
20.9 Interactive Prototyping Tools 503
20.9.1 Kinds of Prototyping Tools 503
20.9.2 Examples of Prototyping Tools 504
20.9.3 Research on Prototyping Tools 506
20.10 Automatic Creation from Models 507
20.11 Tools for 3D, AR/VR/XR Interactions 510
20.12 Models and Patterns for the Implementation 510
20.12.1 Early Models 511
20.12.2 Event Model 511
20.12.3 Listener Pattern 513
20.12.4 Interactors Model 514
20.12.5 Imaging Model 515
20.12.5.1 Retained Object Model 518
20.12.6 Model–View–Controller 519
20.12.6.1 Model–View 520
20.12.7 Document Model 521
20.13 Considerations for the Implementation 521
20.14 Chapter Conclusions 523

Chapter 21 Methods for Designing and Evaluating Interaction Techniques 525

21.1 Introduction 525
21.2 Methods for Needs Finding 525
21.3 Methods for Design 526
21.3.1 Graphic Design 526
21.3.2 Prototyping and Implementing 526
21.4 Methods for Evaluation 527
21.4.1 Usability Evaluation 527
21.4.2 Controlled Experiments 528
21.5 Considerations and Chapter Conclusions 530

Chapter 22 Legal Issues and Patents 531

22.1 Introduction 531
22.2 Copyrights 532
22.3 Trademarks 533
22.4 Trade Secrets 534
22.5 Patents 534
22.5.1 Patent Controversies 537
22.6 Considerations for Intellectual Property and Chapter Conclusions 538

PART IV INTERVIEWS 541

Chapter 23 Interview: Larry Tesler 543

23.1 Short Bio of Larry Tesler 544
23.2 User Studies—Stanford Card Stunt Program 544
23.3 Blank Screen Study 545
23.4 RCG User Study 546
23.5 Cut and Paste 546
23.6 Modeless Software and the Gypsy Text Editor—The Shift Away from Moded Software 547
23.7 The Gypsy Text Editor 548
23.8 Other Problems with Modes 548
23.9 Interaction Techniques at Apple 549

Chapter 24 Interview: David Canfield Smith 551

24.1 Short Bio of David Canfield Smith 552
24.2 The Story of Icons 552
24.3 Pygmalion 553
24.4 Xerox Star 553

24.5 Techniques for Reducing System Commands 555
24.5.1 Universal Commands 555
24.5.2 Creation by Copying 556
24.5.3 Property Sheets 556
24.6 Other Star Designs 557

Chapter 25 Interview: Bill Buxton 559

25.1 Short Bio of Bill Buxton 559
25.2 Interacting with Two Hands 560
25.3 Adjusting Values 561
25.4 Future 561

Chapter 26 Interview: Norm Cox 563

26.1 Short Bio of Norm Cox 563
26.2 Visual Icon Design 564
26.3 Legal Consulting 565
26.4 Interaction Design for Scrolling 566
26.5 Mnemonic Gestures for Interacting with a Device 566

Chapter 27 Interview: Dan Bricklin 569

27.1 Short Bio of Dan Bricklin 569
27.2 VisiCalc 569
27.3 Dan Bricklin's Demo 572
27.4 Chapter Conclusions 573

Chapter 28 Interview: Bill Atkinson 575

28.1 Short Bio of Bill Atkinson 575
28.2 Initial Designs of the Lisa 576
28.3 HyperCard 583
28.4 General Magic 584

Chapter 29 Interview: Bruce Horn 585

29.1 Short Bio of Bruce Horn 585
29.2 PARC and Smalltalk 586
29.3 Apple Macintosh 587

Chapter 30 Interview: Rob Haitani 591

30.1 Short Bio of Rob Haitani 591
30.2 Casio Zoomer 591
30.3 Palm Pilot 592

Chapter 31 Interview: Tandy Trower 597

31.1 Short Bio of Tandy Trower 597
31.2 Precursors to Microsoft Windows 598
31.3 Microsoft Windows 1.0, 2.0, and 3.0 598
31.4 Graphic Design 601
31.5 Microsoft's User Interface Design Group and Visual Basic 602
31.6 Windows NT and Windows 95 603
31.7 Microsoft Bob and the Windows Assistant 604
31.8 Other Microsoft Products 605

Chapter 32 Interview: Loren Brichter 609

32.1 Short Bio of Loren Brichter 609
32.2 Pull-to-Refresh 610
32.3 Other Spring–Action Interactions 614
32.4 Pull-to-Refresh Patent 614

Chapter 33 Interview: Don Hopkins 615

33.1 Short Bio of Don Hopkins 615
33.2 Definitions and Key Properties of Pie Menus 616
33.3 History of Pie Menus 616

Chapter 34 Interview: Ted Selker 619

34.1 Short Bio of Ted Selker 619
34.2 Origins of the Pointing Stick 620
34.3 Refining the TrackPoint 622

Chapter 35 Interview: Shumin Zhai 627

35.1 Short Bio of Shumin Zhai 627
35.2 Effective Text Entry Requirements 628
35.3 Origins of Shape Writing 628
35.4 ShapeWriter 629
35.5 Algorithms 629
35.6 Pixel Squeeze-to-Assist in Google Pixel 2 Phone 630

Chapter 36 Interview: Gregg Vanderheiden 631

36.1 Short Bio of Gregg Vanderheiden 631
36.2 Personal History 632
36.3 Digital Affinity 632
36.4 Morphic 634

36.5 Accessibility Today and Tomorrow 635
36.6 Info-bot 636
36.7 Chapter Conclusions 636

Chapter 37 Interview: Chris Harrison 637

37.1 Short Bio of Chris Harrison 637
37.2 Context 638
37.3 Using the Body for Input/Output 638
37.4 Increasing the Richness of Touch Interfaces 639

Chapter 38 Interview: Julia Schwarz 641

38.1 Short Bio of Julia Schwarz 641
38.2 Devices 642
38.3 HoloLens 2 Interaction Model 642
38.4 HoloLens 2 Prototypes and Interactions 643
38.5 Mixed Reality Toolkit 646
38.6 Chapter Conclusions 647

PART V FUTURE WORK AND CONCLUSIONS 649

Chapter 39 Future of Interaction Techniques 651

39.1 IxT Class Projects 651
39.1.1 New Smartphone Interaction Techniques 651
39.1.2 New Smartwatch Interaction Techniques 652
39.1.3 AR, VR, and 3D Interaction Techniques 652
39.1.4 Wikipedia Entries 653
39.1.5 Comparison and Testing of Historical and Current Interaction Techniques 653
39.1.6 New Testing Methods 654
39.1.7 Parts of This Book 654
39.2 Other Important Future Work 654
39.2.1 UIs for Conversational Agents 654
39.2.2 Better Text Entry 655
39.2.3 IxTs for 3D and Virtual and Augmented Reality 655
39.2.4 Two-Handed Interfaces 655
39.2.5 Brain Interfaces 655
39.2.6 Other GUI IxTs 656
39.2.7 Open Data Model 656

Chapter 40 Conclusions 657

Bibliography 659

Index 703

Author's Biography 741

List of Figures

1.1 A checkbox that is “checked.” 1
1.2 This checkbox shows a square to indicate it is neither checked nor unchecked since both strikethrough and not strikethrough text are selected. 1
1.3 Checkbox showing the (a) blue mouse hover, (b) mouse pressed down, and (c) text focus outline. 2
1.4 This refresh icon was formerly used frequently on smartphones—for example, to check for new mail or new Facebook messages—but has been mostly replaced by “pull-to-refresh,” discussed in Chapter 32. 3
1.5 Some examples of input techniques on computer or smartphone screens. 4
1.6 Some examples of physical interaction techniques for consumer electronics and games. 4
1.7 Simple state diagram for a one-button mouse. 8
1.8 Some controls in the menu for Microsoft Visual Studio. 10
1.9 Picture of me in front of the Alto computer at Xerox PARC in 1979. I sent this annotated picture to my father (in a physical mail letter!) since people were not familiar with the technologies back then. See also Chapter 2. 12
1.10 Many looks for scrollbars over the years. 17

2.1 The ENIAC from 1945 [Rose 1946]. 22
2.2 Punched cards I used at MIT in the 1970s. 23
2.3 A teletype machine with a paper tape reader and punch from the 1960s. 23
2.4 Paper tape I used in high school in 1975. 24

2.5	PERQ computer, monitor, and keyboard. It also had a pointing device. This was the first model of the PERQ, dating from approximately 1981.	25
2.6	Cathode-ray tube.	27
2.7	DEC VT100 display terminal.	27
2.8	Ivan Sutherland's Sketchpad system from 1963 being demonstrated by Timothy Johnson, who created the 3D extension to Sketchpad [Johnson 1963].	29
2.9	(a) The original wooden mouse. (b) The keypad, keyboard, and mouse in use in NLS.	30
2.10	(a) A Smalltalk screen. (b) A close up of two windows, with multiple panes, a scrollbar (on the left), and a popup menu.	32
2.11	A screen showing two panes from the Bravo text editor.	33
2.12	The Xerox Alto (a) Markup and (b) Draw screens.	34
2.13	The SDMS. (a) The overall configuration. (b) A close up of an overview ("world view") screen.	36
2.14	The Xerox Star (a) monitor, keyboard, and mouse and (b) workstation. See also details of the keyboard in Figures 9.1 and 15.1.	37
2.15	(a) The Apple Lisa, and (b) its mouse.	39
2.16	Original 1984 Macintosh.	39
2.17	Microsoft Windows 1.0.	41
2.18	Apple HyperCard, 1987.	41
2.19	The Alternate Reality Kit showing the hand as the mouse cursor, and various objects that could be manipulated.	42
2.20	NeXTSTEP window system for the NeXT computer from around 1989.	43
2.21	Portion of an OpenLook screen.	45
2.22	Portion of a Motif screen.	46
2.23	Approximate timelines showing where and when work was performed on some major technologies.	49
2.24	A command line interface is still available in Windows 10 in the "cmd" window. Here, the user has typed the commands "chdir" (change directory), "dir" (list contents of the current directory), and "ren" to rename a file.	51
3.1	The rise and fall of Windows computers.	59
3.2	Most computing platforms shipped have been mobile since 2010.	60

3.3 HP Jornada 720 handheld PC, from about 2000. The size is 7.40in by 3.70in by 1.30in. 61
3.4 AV touchpad on a 2000s-era Sony Vaio laptop, an early example of a modern laptop touchpad. 62
3.5 Several styluses: (L to R) PalmPilot Professional, Fossil Wrist PDA, Nokia 770, Audiovox XV6600, HP Jornada 520, Sharp Zaurus 5500, and Fujitsu Lifebook P-1032. 62
3.6 HP-65 calculator—first programmable calculator. 64
3.7 Xerox PARC's tabs, pads, and board-size devices. 65
3.8 Reference hardware for Go Corporation's PenPoint system. 66
3.9 (a) A page from the PenPoint user's guide showing tabs, icons, a scrollbar, and menus. (b) The "Flick" gesture for scrolling. 67
3.10 Microsoft PenWindows. 68
3.11 Apple Newton (a) MessagePad 100. (b) Demonstration by John Sculley III. 69
3.12 Doonesbury comic strip from August 26, 1993 about the Apple Newton. 69
3.13 (a) General Magic's "Magic Cap" running on (b) Sony MagicLink hardware. 70
3.14 Motorola Razr with keypad. 71
3.15 IBM Simon from 1994. 72
3.16 Nokia 9110 Communicator: (a) closed and (b) open. 72
3.17 Lucent WaveLAN PCMCIA format card for laptops. 74
3.18 (a) Original Palm Pilot from about 1996. (b) The Graffiti alphabet for Palm. (c) Palm Pilot. 75
3.19 (a) Palm watch by Fossil. (b) The tiny stylus could be stored in the band. 77
3.20 (a) Kyocera QCP-6035 Palm flip phone. (b) A color Palm Treo. 77
3.21 (a) Hewlett Packard Jornada 540 Color Pocket PC from 1999. (b) HP iPAQ 4150 Pocket PC. 78
3.22 Worldwide smartphone sales (%), 2007–2017. 79
3.23 (a) RIM Blackberry 850 from 1999. (b) Blackberry 7210 (first color model) from 2002. Both have the roller dial highlighted. 81
3.24 (a) TabletPC. (b) Bill Gates announcing that the TabletPC will be big. (c) Handwriting recognition on TabletPC. 82
3.25 Original iPod from 2001. 83
3.26 Original iPhone from 2007, with the physical "home" button at the bottom. 84

3.27 Android phone screen showing three icons at the bottom for back, home, and menu. 85
3.28 First version of the iPad from 2010. 86
3.29 Timeline of some of the systems discussed in Chapters 2 and 3. 87

4.1 Two *cursors* at the same time: the text cursor is between the "r" and "s" letters, and the pointer cursor is pointing at the "o." 92
4.2 Various images (a) for a text cursor and (b) for the pointer cursor. 92
4.3 The lightpen. 94
4.4 Arrow keys on (a) a keyboard, (b) a remote control, and (c) Nintendo Switch pro-controller. 95
4.5 Mazda automobile control knob can be rotated to move the selection like step keys or pushed in different directions as a joystick. The semi-circle button on the bottom left is for "back" or "abort." 96
4.6 (a) Rollers on the original SRI mouse. (b) The Xerox mouse used a roller ball and (c) originally had three vertical buttons: red, yellow, and blue. (d) A later Xerox PARC design with three buttons side-by-side. 97
4.7 Mouse gain functions for Windows and Mac, as calculated from measurements and the source code [Casiez et al. 2008]. 98
4.8 (a) Old game joystick. (b) Joysticks on a game controller and (c) on an Epson remote control. (d) Position-controlled joystick. 98
4.9 (a) Kensington trackball. (b) Blackberry Pearl has a tiny trackball below the screen. 99
4.10 IBM TrackPoint joystick and its buttons. 100
4.11 Rand tablet in use. 101
4.12 Digitizer tablet with printed overlay showing specific locations for commands. This must be used in *absolute* tracking mode. 101
4.13 (a) Holding a tablet requires there be an inactive area for the fingers on the front (but notice that this user is still touching the screen with her left thumb). (b) Resting your hand on a tablet can confuse the touchscreen. 103
4.14 Wall and table-size touchscreens. 103
4.15 (a) Small Touch Bar on MacBook is above the keyboard. (b) Four examples of what it may show. 104
4.16 Pointing across the room using (a) the Wii Controllers or (b) finger detection by the Kinect. 104

4.17 (a) Razer Naga 2014 mouse has lots of buttons. (b) Regular mouse with scroll wheel. 107
4.18 My PebblesDraw application, which can support up to eight users at a time, each with their own cursor shape. The user's cursor shape is also used to indicate that user's selected object. 108
4.19 The user is trying to aim a laser pointer at the small square in the center. The beam is turned on at the spot labeled "S," and the yellow line shows the path while trying to get there. Then, the user tries to hold the point as still as possible for three seconds, resulting in the red trace. Finally, the user turns off the beam, which causes the beam to move along the pink path to "E" because the button is on the *side* of the laser pointer, so the pointer moves while the button is being released. 111
4.20 The original apparatus used by Fitts to measure throughput. 112
4.21 (a) The time to position a cursor varies linearly with the distance for step keys but by the log of the distance for continuous devices. (b) The positioning time for both the mouse and the joystick decreases with the log of the target size. 113
4.22 The values calculated for a and b for various devices result in curves that closely match the measured values. 114
4.23 RollerMouse Red *Plus* pointing device from Contour. The bumpy black cylinder (at arrow) is *slid* left and right and *rolled* forward and backward to move the pointer cursor. 114
4.24 Two examples of the circular pointing device speed and accuracy test, with different size and distance of targets. The blue dot target moves across the circle. 115

5.1 Scrolls of parchment date back over 4,500 years. 120
5.2 Scroll wheel properties in Windows 10, set to three lines per increment ("notch"). 122
5.3 Early wearable system used a head-mounted display and a knob on the hip that can be operated without looking at it. 122
5.4 Panels created with the Pebbles Shortcutter program: (a) numeric keypad; (b) collection of scrollers and a knob; (c) buttons for controlling items in a set of compilers; (d) gesture pad and two rows of small buttons; (e) controller for the WinAmp PC program for playing music files; (f) panel for browsing via Internet Explorer; (g) panel that interfaces with X-10 to control room lights; and (h) panel that controls a video projector. 123

5.5 Audio mixing console with 1D sliders at the bottom and knobs above. 124

5.6 (a) WordStar scrolling commands (circled in red) and soft key labels for the F1 to F10 keyboard keys at the bottom (1978). (b) Some labeled keys from a PC keyboard. 125

5.7 (a) Bravo's scrollbar was the white region to the left of the text. When the pointer cursor was moved there, the cursor picture changed to an up–down arrow. (b) The left mouse button scrolled up, (c) the right button scrolled down, and (d) the middle button was thumb, (e) which also can be used to see where you are in the document. 128

5.8 (a) Smalltalk's scrollbar appeared when the cursor was in the window. The gray box in the center shows what part (both position and percent) of the contents is currently visible in the window. When the cursor was on the right side of the scrollbar, pressing the left mouse button scrolled up and (b) the left side scrolled down. (c) Pressing in the center would drag the thumb indicator up and down, with the outline box showing the original position. 129

5.9 (a–d) Interlisp-D scrollbars worked like a combination of Smalltalk and Bravo, using all three mouse buttons. (e) Symbolics vertical and horizontal scrollbars. 130

5.10 Details of Xerox Star (a) window and (b) scrollbar, which also shows the menu or "hamburger" icon (see also Figure 7.2(a)). 132

5.11 Apple Lisa screen from 1983. LisaDraw is shown in the window at the center-right with the title "Peggie's Rose." Its scroll bar is at the right, and its palette on the left has the top arrow selected. The menu bar at the top has "Copy" selected from the "Edit" menu and contains various available and grayed out menu items. 132

5.12 (a) Apple Macintosh screen. The System Folder window has the focus, so it shows the title bar decorations and the scroll arrows, but the contents fit into the window, so the gray thumb is not shown. (b) View of scrollbar from the video showing when content does not fit. 134

5.13 (a) Mac OS 9.0 in 1999 moved the scrollbar increment buttons to the bottom, based on the NeXT design. (b) Mac OS X 10.7 Lion from 2011 eliminated the increment buttons, based on the iPhone design. 135

5.14 (a) OpenLook (1988) put the increment arrows on the thumb, (b) as did Google Wave (2010). (c) NeXT (1989) moved the scrollbar back to the left and both arrow buttons to the bottom. 136
5.15 Palm scrollbar. 137
5.16 (a) Main iPhone scrollbar (highlighted with a red ellipse) is just for feedback and is not clickable. (b) Special scrollbar for alphabetical lists. 138
5.17 (a) The Chrome browser, version 66.0.3359.117 for Windows, displays orange lines in the scrollbar showing where the search term (here "Myers") appears in the document. (b) Similarly, the Eclipse code editor, version: Neon.3 Release (4.6.3), displays red rectangles next to the scrollbar to show where there are compile errors in the current file. (c) Shows a representation of the entire file as the scroll bar, with a box as the part currently viewed. 139
5.18 A picture in MacPaint 1.0, drawn by Susan Kare. 141
5.19 (a) Adobe® Acrobat® Pro DC (Beta 6), with hand scrolling cursor and icon highlighted. (b) Close up of hand cursor open and closed. 142
5.20 The mouse cursor changes to show that vertical or horizontal rate-controlled scrolling is engaged. 143
5.21 (a) US Patent 7,469,381 Fig. 8C shows that when the user pulls in the direction of the arrow labeled 3925, the content moves a shorter distance, between 3928-1 and 3928-2, and will bounce back when the user lifts the finger. (b) The iPhone operates similarly—see the space between the two red arrows as I pull down on my web page in Safari. (c) The Android default browser displayed a transparent elliptical overlay instead. 145
5.22 (a) and (c) Various sliders. (b) Slider on iPhone v11.3.1. (d) A current two-indicator range slider specifying the desired time range. 147
5.23 Video playback indicator at the bottom shows the current position (59:26) and a thumbnail of the video at the pointer position (12:50), and it can be used to monitor and adjust where it is playing. 148
5.24 Tasks for scrolling to a known location. Prompt at the bottom told users how far they needed to scroll and which side of the screen it would be on. 149
5.25 Using a PDA in the left hand to scroll while using the mouse in the right hand. 150

5.26 The user interface of ScrollTest contains two main pages: (a) Scrolling tests with *unknown* position target and (b) scrolling tests with *known* position target. Both pages have (1) instruction panel, (2) frame where the target line should be scrolled to, (3) target line, and (4) scrollable area. 151

5.27 (Top) Average movement time across all conditions (shorter is better) for all 11 devices tested using ScrollTest. (Bottom) Regression analysis of average movement time and distance across all devices—showing that the time was mostly linear when the target position was unknown (a), but matched a log scale like Fitts law when the target position was known (b). 152

6.1 Pictures in the pointer cursor to show the computer is busy. (a) Static hourglass from the Xerox Alto and Star. (b) A Buddha cursor from the University of Toronto to signify patience is necessary. (c) The watch from the original Macintosh (1984). (d) Dynamic cursor from Alto FTP program. 158

6.2 (a) A count-down light clock from 1976. (b) The original Macintosh MacTerminal progress bar. 159

6.3 Three icons from the Sapphire window manager. (a) Random progress. (b) and (c) Regular progress bars with two or one level. 159

6.4 Two screenshots of the progress dialog in Windows XP. The blue and white paper repeatedly flies from the left folder to the right one. This design dates back to at least Windows 95 (1995). 161

6.5 (a) Windows 10 progress dialog. (b) Windows 7 progress in folder field—the green shading fills from the left to the right end of the field. (c) Windows 10 taskbar progress. 161

7.1 (a) Checkboxes, (b) radio buttons, (c) font style submenu popped up from a combo box, and (d) clipboard menu in Microsoft Word 2016 (part of the Office Ribbon). 164

7.2 (a) Xerox Star menu button and its menu. (b and c) Original Macintosh combo box before and after pressing on the button. (d) A combo box supporting text entry, a menu with a scroll bar, and a "Recently Used Fonts" section. (e) Menu from Microsoft Word for Mac with scroll arrows at top and bottom. 166

7.3 Fixed textual menu. User makes a selection by typing on the keyboard. 170

7.4 Examples of "cascading" submenus: (a) submenus in Smalltalk can be arbitrarily deep. However, (b) the original Macintosh submenus are recommended to be only one level deep. 173

7.5 Menus in a recent version of Adobe® Acrobat® Pro DC (Beta 6). The top-level menu selection is shown with a blue background and the text changes color to be white, whereas submenus use a light blue background, and the text does not change color. 174

7.6 Original Macintosh menu bar with selected options in a submenu shown with a checkmark. 174

7.7 (a) Menus on the Palm Pilot. (b) Spin menu on iPhone iOS v.3. 175

7.8 Xerox Star property sheet. 176

7.9 Various checkboxes. 177

7.10 Various radio buttons. 177

7.11 Research on different ways for toggle switches to appear on a touchscreen. 178

7.12 Original Macintosh Cancel–OK buttons. The thicker border shows that OK is the default. 178

7.13 (a) Pie menu. (b) Flexible control. (c) Too many items. 180

7.14 Projecting menus on the hand and arm. 181

7.15 Converting a popup into a fixed menu. (a) HyperCard tear-off before and (b) after. (c) OpenLook had a pin icon at the upper left of some menus. 184

7.16 Windows 2.0 menu showing the underlined mnemonics (e.g., "M" for Move and "x" for Maximize) and shortcuts (e.g., Alt+F7 for Move). 185

7.17 (a) Xerox XDE stacked menus. (b) The Andrew menu stack. 186

7.18 Microsoft Office XP "personalized menus" showed (a) the most frequent or (b) all commands. 189

8.1 Sholes typewriter, 1873. 194

8.2 Composition of Chinese and English writing from the smallest units. 197

8.3 The pinyin system. 198

8.4 A comparison of Chinese characters in Zhuyin and pinyin forms. 199

8.5 MacOS Chinese pinyin text entry popup. 200

8.6 iOS 15 pinyin input interface status after typing. 200

8.7 (a) and (b) Pages from the PenPoint user's guide showing the hand printing boxes. 202

8.8 Unistrokes gestures for each character and the median time in milliseconds spent drawing each unistroke. 204
8.9 (a) With EdgeWrite, users move along edges and to the corners of a plastic guide with a square hole mounted on a Palm PDA, to provide additional stability and accuracy. (b) The EdgeWrite gestural alphabet. 205
8.10 (a) Dial telephone showing letters associated with the numbers. (b) Conventional touch-tone telephone from AT&T. (c) Close up of keypad for a mobile phone. 206
8.11 Versions of the Twiddler (a) used in the 2004 research study [Lyons et al. 2004] and (b) in use by Thad Starner with a heads-up display. 209
8.12 Virtual keyboard displayed by the Xfinity (Comcast) set-top box for searching. The user moves the character selected using the remote's arrow keys. 210
8.13 Virtual symbol keyboards (a) on Xerox Star from 1981 and (b) on Microsoft Office 2016. 210
8.14 (a) The on-screen keyboard from the Apple Newton would pop up when the user tapped the keyboard icon at the bottom. (b) Palm's virtual keyboard for letters. 211
8.15 (a) The iPhone shows the current key in a popup since the key is hidden by the finger. (b) Some iPhone keys pop up alternatives that the user can select. 212
8.16 (a) The OPTI optimized virtual keyboard. (b) Metropolis virtual keyboard. 213
8.17 (a) Entering the word "the" with Quikwriting: the user started at the dot, moved to "t" and back, then moved to the area for "h", which was at the top of that area so the user moved from that area upward, followed by re-entering that area for "e". (b) Entering the word "finished" with Cirrin required moving the pointer into and out of each desired letter. 214
8.18 (a) ShapeWriter. (b) Swype. 215
8.19 (a) ZoomBoard zooms into the part of the keyboard where the user taps, so the keys get to be accurately tappable. (b) Watchboard prototype text entry using physical keys that might slide out from a watchband. 216
8.20 Accelerometers on each finger of the Tap Systems device allow text entry by tapping chords on any surface. Moving the thumb along a surface moves the pointer like a mouse. 217

8.21 (a) Spell checker in Microsoft Word 2003. (b) Auto-correct in Microsoft Word. 219
8.22 Wavy underlines: (a) red for spelling and (b) green for grammar. (c) Word 365 still uses red wavy underlines for spelling but switched to double blue underlines for grammar. 220
8.23 Wavy underline in the Eclipse IDE, with auto-correct menu of quick fixes. Clicking on the red "x" icon in the left margin also pops up the quick fixes menu. Also shown are the different colors for the code elements. 220
8.24 Visual Studio pops up a menu of autocompletions after the user types a ".". 221
8.25 The AA.com website provides a list of autocomplete options in the destination field. 222
8.26 Chrome search bar (address bar) autocomplete. 223
8.27 (a) POBox autocomplete—when the stylus touches the "e" key, it pops up likely words and the user can slide to select or release to pick only the letter. (b) TextPlus autocomplete for Palm from SmartCell Technology in 2000. 224
8.28 (a) Original iPhone's auto-correct and complete suggestions were shown in a popup and used by default. (b) Since iOS 8, three suggestions are shown above the keyboard (this picture is from iOS v. 12.4.1). (c) Misspellings have been marked with a dotted underline, which when clicked display corrections. 225
8.29 SwiftKey autocomplete product from 2010 displays three words on Android. 225
8.30 Example of auto-correct changing the content in unexpected ways. 226
8.31 ListPad uses web services to provide context-dependent autocompletes, such as the sessions' names for a conference (at 3). 227
8.32 The learning curve for the EdgeWrite text entry method (Section 8.3.2) goes *down* since it measures time. 229
8.33 Formatting for a typewriter-like printing from TJ-2 in 1963. 234
8.34 Smalltalk Browser for editing code—the top row shows the (from left to right) categories, classes, method categories, and methods, and the bottom pane shows the code for the selected method. 237
8.35 (a) On character terminals, the text cursor is *on* a character (here "m"). (b) and (c) The Bravo text editor showed the selected text by underlining, here a capital "A" in (b) and the word "complete" in (c). (d) Smalltalk's text cursor was a caret shown *between* characters. 239

8.36 Vertical movement of the cursor when (a) there are multiple fonts or (b) at the end of the line. 240
8.37 (a) Pressing on text on older versions of the iPhone displayed the text around the finger in a popup magnifying glass (also called a *loupe*). (b) Tapping twice in the same spot brings up the select buttons. (c) The handles for controlling the selection and the popup copy button. (d) iOS 15 in 2021 brought back the magnifying glass as a rounded rectangle. 242
8.38 (a) Single line and (b) multi-line text widgets from web pages. (c) A spinner to the right of a text field for entering numbers. 246

9.1 Special keys on the Xerox Star keyboard, including the "PROP'S" key at the bottom right of the left group. See also Figure 15.1 for the right-side special keys. 250
9.2 Detail of top of the property sheet shown in Figure 7.8. 251
9.3 Dialog boxes on Apple Macintosh that are (a) modal and (b) not modal. 251
9.4 (a) Symbolics dialog with scroll bar. (b) The Fabrik system from 1988 grouped elements into different tabs. 252
9.5 (a) Modern Windows tabbed print dialog for Word 2016 in Windows 10. (b) Adobe's patent that describes draggable tabs. 253
9.6 (a) Four rows of toolbars from Win SCP version 5.21.1. (b) The ribbon in Microsoft Word 365 version 2207 Build 16.0.15427.20182. 254
9.7 (a) Palm Pilot page of settings and (b) popup dialog box. (c) iPhone iOS v. 13.2 settings screen. 256
9.8 Jade automatically generated a dialog using the (a) Garnet style and the (b) OpenLook style for specifying the properties of a text object. (c) Control panel for a copier automatically created by PUC for a PDA. 257
9.9 The print dialog for Windows in (a) English and (b) German. Note that the "pages" text is smaller and the buttons at the top right are wider. 259

10.1 (a) The bounding box of the green line is the whole dotted rectangle. Clicking at the arrow should cause the gray rectangle to be selected, not the line or the red ellipse, even though both are on top of the square, and also not the blue triangle, which is behind the gray square. (b) The intersection of two lines can define four segments, (c) so each could be individually edited. 265

10.2 (a) These objects are in a group, which is shown by the dotted outline and handles on the outline. The text label is also selected and is editable while still in the group. (b) Selection handles show through objects on top. 266
10.3 (a) Close up of small squares used to show the selection in Xerox Draw from 1975. (b) Selection feedback on the Star from 1982. 267
10.4 Selection in the original Lisa and Macintosh used (a) handles and (b) reverse video. 268
10.5 (a) When a red line is drawn over a red square in a painting program, the integrity of the line is lost. (b) The lasso tool and "marching ants" selection feedback in the original Macintosh MacPaint. 269
10.6 In PowerPoint 2016 (v. 16.0), pressing on the handle shown at the base of an arrow in (a) and dragging in the arrow's direction across the opposite end results in different flipping for (b) graphics versus (c) a text box. 272
10.7 In PowerPoint 2016 (v. 16.0), selecting multiple objects separately (a) and resizing them all using the handle on the bottom right one, (b) resizes each from that object's upper left corner (so they all stay at their original places). However, grouping the objects (c) and resizing the group (d) affects each object's size *and position* proportionally. 273
10.8 Thick lines go on both sides of the origin point. These two squares have the same "Y" coordinate, but the selected one on the right has a thickness that is 5 times the one on the left, so it goes 2 in each direction. 274
10.9 (a) Polyline defined by 4 points. (b) Self-intersecting polygon defined by 5 points. (c) Curved spline from 3 points. (d) Spline control handles. 275
10.10 Gridding causes the edges of objects to exactly *overlap* (a) and *not* be adjacent (b). 277
10.11 Whereas normally (a) the end of the line precisely follows the cursor, shown by the "+", (b) when the line is close to an object, it *snaps* away from the cursor to the point on the object, and the snapping points on the object are highlighted. (c) After snapping, the connection is retained as a *constraint* if either object moves or changes size. (d) When dragging the yellow square, it snaps to "smart guides" (dotted red lines) based on the other objects in the vicinity. 277

10.12 Widgets and dropdown dialogs for selecting the text (a) color and (b) background color, and the widgets for a graphical object's (c) fill and outline. 278
10.13 Color chooser widgets. (a) Built-in for Java Platform SE 7. The (b) standard and (c) custom "More Colors" dialogs from Word and PowerPoint 2016 (v. 16.0). 279
10.14 (a) A texture pattern can be anchored to the window so multiple objects match up, or (b) anchored to objects so it stays consistent when the object is moved, but then there may be seams between objects. 280
10.15 Hierarchy views. (a) Generic. (b) File Explorer in Windows 10. (c) Eclipse editor for Java version Neon.3 Release (4.6.3). 283
10.16 Electronic programming guide as a 2D grid. 284
10.17 Gargoyle editor from Xerox PARC, selecting (a) the lower left endpoint with a single click, then (b) clicking again selects the segment, and (c) clicking a third time selects the whole object. 287
10.18 Alternate Reality Kit showing the cursor as a hand and a slider (a) before and (b) after the slider is selected and picked up. 287
10.19 The Topaz research graphical editor showing (a) the search dialog and (b) the result of replacing a triangle with three triangles, applied recursively multiple times. 288
10.20 The properties of the drawing of the seagull are represented in the cards on the right, which can be easily manipulated using one or more fingers. 290

11.1 A screen from the Sapphire window manager from 1984. The icons are collected into a special window that by default is at the bottom of the screen. W2 has the keyboard focus, as shown by its thicker border, which is also shown on its icon ("ShowSc"). Progress bars (Chapter 6) are shown in the title bar of each window and in most icons (see also Figure 6.3). 299
11.2 The Program Manager window in Windows 3.0 (1990) uses MDI to show the Main and Accessories windows inside of it, each of which can be moved around and changed size, but must stay inside the Program Manager window. 300
11.3 A screen from the Cedar environment from Xerox PARC, showing its tiled windows and two rows of icons at the bottom. Most windows are displaying a row of buttons just below the title line. 301

11.4 Tiled windows can be in arbitrary places (left) or in fixed columns (right). 302
11.5 (a) Tabs for web pages get tiny when there are too many open in Chrome v. 79.0.3945.130. (b) Excel 2016 provides arrows and ellipses when there are too many tabs. 304
11.6 Many icons: (a) The icon for the Microsoft Word program. (b) Icons on the Apple Watch for various applications. (c) Three icons on Windows for folders and a file. (d) Icons on the iPhone—the bottom row is the iPhone dock or bottom bar, and the row above it contains two applications, then the Games folder, and then an icon for my web page. (e) Windows Taskbar showing running applications. 305
11.7 Icons from various historical systems: Sapphire [Myers 1984], original Macintosh [Williams 1984], X [Scheifler and Gettys 1986], Cedar, [Teitelman 1985], and Star [Smith et al. 1982b]. Some of the X icons contain the actual text displayed in the window in a tiny (unreadable) font. 306
11.8 Palm handheld used icons to represent all applications that could be run. The four physical buttons at the bottom and the calculator icon next to the text input area also start applications. 307
11.9 (a) Windows 95 showing icons on the left and the taskbar at the bottom with the “Start” button. (b) NeXT computer’s Dock. (c) Macintosh dock as it appeared in OS X 10.8 to 10.9. 308
11.10 Two “Rooms”—top for email and bottom for programming. 314
11.11 Lifestreams. 318
11.12 TimeScape showing (a) the timeline view and (b) zoomed in for finer details. 319
11.13 Microsoft Bob. 320
11.14 Feldspar allows finding items by association. 322

12.1 Researchers with the European Space Agency in Darmstadt, Germany, equipped with a VR head-mounted display and motion controllers. 326
12.2 Google Glass superimposes an image that users see with their right eye. The side of the device is a touchpad. 327
12.3 Google Cardboard holds a smartphone (here visible at the right in yellow) in front of one’s eyes. A hole in the back allows the phone’s camera to work for AR uses. 328

12.4 *Pokémon Go* augmented reality (AR) game. The green character and red "Poké Ball" are superimposed over the camera view of the machinery and ground. 329
12.5 The six degrees of freedom (6DOF) for 3D objects. 330
12.6 Example spatial reasoning test questions. 331
12.7 Mechanical apparatus for tracking the user's head from 1968, colloquially called the *Sword of Damocles*. 333
12.8 Data glove with outer covering removed to show sensors. 334
12.9 (a) XWand. (b) Nintendo Wii Remote. 335
12.10 (a) Kinect for Xbox 360. (b) A handheld projector with attached balls as markers for tracking. (c) Leap Motion sensor. 337
12.11 (a) Phantom Omni. (b) Novint's Falcon. 338
12.12 NOOBIE stuffed animal with pressure sensors and a display on its stomach. 339
12.13 Articulated "marionette" with sensors in each joint. 341
12.14 (a) Atari *Pong*. (b) Namco *Pac-Man*. (c) Atari *Centipede*. (d) *Dance Dance Revolution*. 342
12.15 Distribution of the US$134.9 billion game industry in 2018. 344
12.16 (a) Constructive Solid Geometry (CSG) creates complex shapes through union and intersection of basic shapes. (b) Realistic trees created by procedural generation, controlled by changing parameters and a random number. 345
12.17 Four views of a 3D object—along each of three axes and perspective. 346
12.18 In Teddy, the user draws fluid curves to define 3D shapes. 347
12.19 3D manipulations for a VR environment with tracked hand and fingers. (a) A third person view of the Head Crusher technique. The inset shows the first person view. (b) A third person point of view of the Lifting Palm technique. The inset shows the first person view. 348
12.20 Handles for 3D manipulations of simple objects. (a) The starting configuration for a pink cube and a rack with twist (purple), taper (blue), and bend (red) handles. (b) The taper handle is translated downward, tapering the cube. The deformation range is the region between the twist and taper handles. 349
12.21 SketchUp 3D handles for scaling. 350

13.1 (a) LOGO Turtle controlled by buttons. (b) First-generation RCX programmable brick. 356

13.2 (a) Furby from Tiger Electronics. (b) Sony AIBO dog. (c) Aldebaran Robotics Nao. 357
13.3 3D printed pneumatic button (A), knob (B), and slider (C). 359
13.4 Phidgets controlling physical objects made by students: (a) Eyes closed, partially open and lit, and fully opened. (b) Waterfall harp that responds to the user's finger position using light sensors. 360
13.5 Two people each with a handheld projector, exploring an information space on the wall together. 362

14.1 Copy-editing marks for proofreading. 371
14.2 SILK recognized rectangles, circles, squiggles (as text), and other shapes. Combinations are recognized as other objects, such as two rectangles as a scroll bar (left), a row of squiggles as a menu bar (top), and squiggles in ovals as buttons (bottom). 373
14.3 The ASSIST system recognizes sketches as physical items and runs a simulation. 374
14.4 (a) Rubine's training tool with examples of a single-stroke "X" gesture. (b) The Agate tool from Amulet, which uses Rubine's algorithm, with a gesture set (top) and a new gesture being demonstrated (bottom). 375
14.5 Gestures used to test the $1 recognizer. 377
14.6 (a) Shows the authors wearing toy noses—I am in the bottom left. (b) Rotating with one's nose on a CRT display. 379
14.7 Taxonomy of surface gestures. 380

15.1 Special keys on the right side of the Xerox Star keyboard, including the "UNDO" key. See also Figure 9.1 for the rest of the keyboard. 387
15.2 iPhone iOS 13.6 multi-level undo in Notes: I did style-change, then text edit, then undo (of the text edit), and when I shake the device again, it offers to redo the text edit (undo the undo) or else to undo the previous style change. 388
15.3 Example of linear undo. 395
15.4 (a) Office 2016 undo stack. (b) Adobe® Photoshop® 2017 History Panel (far right). 396
15.5 Example of Emacs undo. 397
15.6 Example of Emac editor displaying the Tree Undo visualization. The dark path on the left is what is in effect now, and the lighter paths are available. 398

15.7 Selective undo. 399
15.8 Chimera script model for undo/redo—adding a new "Close Trajs" operation in the past. In the graphical History Panel, each vertical pair shows the "before" and "after" of each operation. (a) The original history, and (b) after editing the past to make the eye gray. Subsequent operations, like drawing the mouth, are redone. 400
15.9 Aquamarine's History Panel showing operation #10 (brush stroke for the nose) selectively undone. 401
15.10 (a) In Topaz, the user can select operations (which do not have to be contiguous). to be part of a macro. (b) Executing the Sierpiñski Gasket macro 10 times. (c) A macro that converts an And gate and a Not gate into two Not gates and an Or gate and reconnects the wires (De Morgan's law). 402
15.11 Azurite dialog that shows the *diff* between the current code (left) and the code in that region at a selected time in the past. 402
15.12 Example of selective undo using the inverse model. 404

16.1 Microsoft Word's status line while saving this book. 414
16.2 (a) Balloon Help on the Macintosh. (b) Tooltips in Microsoft Office. 416
16.3 In Eclipse, tooltips that appear when users (a) hover over an item can be turned into (b) regular windows where users can scroll and click on items. 416
16.4 Clicking on the 5th icon in LabVIEW (detail at right) goes into help mode. 418
16.5 PowerPoint 1995 help menu, containing "What's this?". 418
16.6 (a) When the user asks about a location in Crystal, the system goes into a mode (b), where the next click drops a pink question mark (c) and pops up a menu of automatically generated questions about what happened there. 419
16.7 Blue Stencil covers the screen except for the "trevor" button that the user should click on. The pink text at the top tells the tutorial step, and the yellow note explains what to do in this step. 420

17.1 Stephen Hawking, a noted physicist with ALS who used an augmentative and alternative communication (AAC) device. 426
17.2 The Kensington Orbit® Trackball with Scroll Ring that I used to write this book. Note that one can use different fingers to move the ball and gray scroll ring, and the thumb is used to press the left button, which allows the index finger not to be used. 428

17.3 Dan was a 27-year-old with muscular dystrophy in 2001 who typed using two pencils and used a trackball (on the pink wrist pad) for pointing. 430
17.4 Early 1960s picture of user operating an electric communication device in the form of a sip-and-puff typewriter controller named the Patient Operated Selector Mechanism (POSM or POSSUM), which was developed in the United Kingdom. 431
17.5 (a) Red cap contains EEG electrodes. (b) Monkey at right has implanted electrodes in the brain and has learned to control the robotic arm. 433
17.6 Microsoft Windows 10 “Ease of Access” Control Panel and Magnifier control. 436
17.7 Some of the many screens of controls for the iPhone’s accessibility features in version 14.4. 437

18.1 (a) Version of the Apple Siri screen from around 2013 showing prompts. (b) 2015 version adds prompts for various apps. 445
18.2 (a) Three places to initiate listening on the current iPhone Messages app—the power button on the side for commands, the icon in text field to record an audio message, and the icon on the keyboard to start transcriptions for dictating the message body. (b) While dictating, the user can be silent or press the keyboard icon to stop. 446
18.3 The interface of Sovite: (a) Sovite shows an app GUI screenshot to communicate its state of understanding. The yellow highlight overlay specifies the task slot value. The user can drag the overlay to fix slot value errors. (b) To fix intent detection errors, the user can refer to an app that represents their desired task. Sovite will match the utterance to an app on the phone (with its icon shown) and look for intents that use or are relevant to this app. (c) If the intent is still ambiguous after referring to an app, the user can show a specific app screen relevant to the desired task. 449
18.4 Put-That-There combined speech with pointing gestures as an early multi-modal interface. 450
18.5 In Quickset, the user could speak (here: “Pan”) and gesture (the yellow arrow) and the system would combine them to recognize what to do (here, pan down). 451
18.6 The intelligent agent in Apple’s 1987 concept video is shown with a bow tie in a window. 452

18.7 Microsoft Clippy. 453

18.8 The “Baxter” robot from Rethink Robotics from 2015–2018 displayed a face on its screen. 455

18.9 (a) When the first two items of the list (shown at the top as “Items = ”) are used in the Drawing Area (center), then (b) Peridot proposes to instead use an iteration using all the items, asking the user with a question-and-answer interaction style (bottom). 457

19.1 A simple state transition diagram with two states and two transitions. 469

19.2 Somewhat more complete state diagram for a control such as a checkbox. 470

19.3 Buxton’s 1983 taxonomy of characteristics of input devices. 473

19.4 Card, Mackinlay, Robertson Design Space (1990). The shapes reference the Foley & Wallace (Section 19.5.1) and Buxton (Section 19.5.2) classifications. 474

19.5 Memories and processors of the Model Human Processor and their time constants. 477

19.6 Times to perform an editing task as predicted by KLM for a particular editor. **H** is the time to home to the mouse or to the keyboard. **P** is the time to point. **K** is the time to hit a keyboard key or a mouse button (YELLOW was the middle mouse button), where 5**K** means 5 keys. **M** is the mental operation (here followed by a K operator). 478

19.7 CogTool. At left are some of the screens (“Frames”) of the Eclipse application, with arrows (“Transitions”) showing what the user does to perform a particular task. 479

20.1 Layers of UI software tools. 486

20.2 The stretchiness of the glue objects in InterViews, represented by double-headed arrows, controls the horizontal and vertical placement of the message and button objects. 496

20.3 The “Print” dialog box in Microsoft Word in different languages. 497

20.4 The “Font” dialog box in Microsoft Word in different languages. Note the extra tab in the German one (circled at right). 498

20.5 Screenshot of the Visual Basic system, with the palette of widgets on the far left, the form editor at the upper center where the user is laying out widgets, the property sheet for the selected widget (here, the whole Elevator Simulation dialog box is selected), and the code editor at the center bottom. 500
20.6 The OPUS editor allows the user to place *constraints* as lines with arrowheads among the widgets and references lines (without arrowheads). 501
20.7 The user can show the look and behavior of widgets by example with Peridot. (a) Starting from the menu shown in Figure 18.9, the user can demonstrate how the highlight moves with the mouse. (b) Similarly, the various stages of drawing what a scroll bar looks like with rectangles, followed by demonstrating how the gray indicator bar should move with the mouse's middle button. 502
20.8 InterState's transition diagram with two states: idle and dragging, with transitions due to mousedown or mouseup. The current state controls which spreadsheet columns are in effect for the properties at left. Here, the X,Y properties are using the initial values since it is idle, but if it switches to dragging, then the mouse.x and .y will be used instead. 503
20.9 The behavior menu from Adobe® Flash® Catalyst 5.5. 504
20.10 Example of a wireframe prototype. 505
20.11 Storyboard in SILK shows transitions for buttons (left) that pop up a dialog box and (right) that make it go away. 507
20.12 User interfaces generated by PUC for a complex and simple copier without and with consistency: (a) complex copier without consistency, (b) simple copier without consistency, (c) complex copier consistent with simple copier, and (d) simple copier consistent with complex copier. 509
20.13 Seeheim model. 511

21.1 Nielsen's 10 heuristics. 529

22.1 Icons representing how to delete files, from (a) the original Apple Macintosh and (b) Microsoft Windows. 532
22.2 Cover page of Jacob Wobbrock and my patent. 535
22.3 Number of software patents issued. 538

23.1 Brad Myers giving Larry Tesler the "SIGCHI Lifetime Practice Award" in 2011. 543

23.2 Stanford students lift appropriate cards to make a red and white letter "S." 545
23.3 Larry Tesler's license plate was "NOMODES." 549

24.1 David Canfield Smith (right) giving a demonstration to Brad Myers (left) of the KidSim system [Smith et al. 1994] at CHI'1995, which eventually became Stagecast Creator. 551
24.2 Final screen from demonstration of factorial using Pygmalion. 554

25.1 Bill Buxton (left) with Brad Myers at the 2014 SIGCHI conference award banquet. 559
25.2 Music system from 1978. From left—physical buttons and sliders, character terminal with columns of numbers that can be selected and set using the keyboard or pointer; piano keyboard; digitizer with puck pointing device with four colored buttons. 561

26.1 Norm Cox. 563
26.2 (a) Original and (b) final design for the printer icon. 565

27.1 Dan Bricklin. 569
27.2 Early plan (Fall 1978) for the operations and modes of the spreadsheet, and keystrokes to invoke them. 570
27.3 Still from Dan Bricklin's demo of VisiCalc running on an emulator of VisiCalc 1981, showing a formula for D4 (at the top) and two panes of cells. 571
27.4 Still from Dan Bricklin's demo of Dan Bricklin's Demo II running on an emulator. 572

28.1 (a) Steve Jobs and Bill Atkinson (right), taken by Norman Seeff in January 1984. (b) Screen capture from 2019. 575
28.2 (a) Initial ideas for softkeys and white on black. (b) Black on white text, with "Thing & Place" interaction, and initial menu bar at the bottom. 577
28.3 (a) Early scrollbar experiment on the left and the "Waste Basket" visible at the bottom. (b) Initial menu bar design, with the resize icon in the menu bar. 579
28.4 (a) Updated menu bar, scrollbars to the right and bottom, grow icon at the bottom right. (b) Early design for tool palette that supports drawing modes. 580

29.1 (a) From left: Randy Wigginton, Jerome Coonen, Rony Sebok, Donn Denman, Andy Hertzfeld, Bruce Horn, Bill Atkinson, Susan Kare, Owen Densmore, Steve Capps, Larry Kenyon, Patti Kenyon, Tracy Kenyon, Steve Jobs, taken by Norman Seeff in January 1984. (b) Bruce Horn more recently. 585

30.1 Rob Haitani. 591
30.2 Haitani's comparison of the Zoomer device vs. the Apple Newton (pre-Palm Pilot). 592
30.3 Original Palm calendar screens. 593
30.4 Iterations of the screen designs. 594

31.1 Tandy Trower. 597
31.2 Microsoft Interface Manager screens from 1983. 598
31.3 Cedar environment from Xerox PARC, which Scott McGregor worked on. 599
31.4 Windows 1.0 Premiere Edition released in August 1985. 600
31.5 IBM's TopView window manager. 601
31.6 IBM and Microsoft worked together on OS/2 Presentation Manager. 602
31.7 Windows 2.0 from 1987. 603
31.8 Windows 95. 604
31.9 Windows Media Center (2002). 605
31.10 Windows 6 Vista (2006). 606
31.11 Microsoft Robotics Studio (2006). 606

32.1 Loren Brichter. 609
32.2 (a) Tweetie main screen. (b) First design for pull-to-refresh. 611
32.3 Pull-to-refresh—first version of the feedback (stitched together from independent screenshots, which is why the dates are different). 612
32.4 Apple's original pull-to-refresh. 613

33.1 Don Hopkins. 615
33.2 Clicking on a character in *The Sims* popped up a dynamic pie menu, here with eight items, and the character's face in the center. 617

34.1 Ted Selker. 619
34.2 Gain function used on an early pointing stick in 1990. 621
34.3 Original pointing stick design was a rod. 622
34.4 Experimenting with using two TrackPoints. 624

35.1 Shumin Zhai. 627
35.2 1,000 randomly sampled taps on the “F” key shows that many taps are on neighboring keys, many of which can be corrected by using a language model. 630

36.1 Gregg Vanderheiden. 631
36.2 A screenshot of the Morphic accessibility settings from Vanderheiden’s presentation slides. 634

37.1 Chris Harrison. 637
37.2 Mounting a projector and sensors onto a person’s body. 638

38.1 Julia Schwarz wearing a HoloLens 2. 641
38.2 First prototype—pinching the corners of a cube. 643
38.3 Three ways to rotate—pinch the purple handle, poke the indentation with a finger, or grab the whole knob with the hand. 644
38.4 Flying hands to interact at a distance. 646

List of Tables

3.1 Four mobile phone generations 73

8.1 Speeds for various text entry methods in this chapter. The wpm are given for different levels 196

Acknowledgments

I started writing this book in 2014, so there are a lot of people to thank. First, I want to thank my wife, Bernita Myers, and family for supporting me through the many years of writing and revisions! I would also like to thank my many mentors and advisors throughout the years.

This book inspired and was inspired by a course I taught multiple times on *Interaction Techniques*, and I want to thank all the students who took the course in the spring semesters of 2014, 2016, 2019, and 2022, especially Bobbie Chen and Wanling Ding from the spring 2019 class for editing, proofreading, transcribing, and making some new pictures for an early version of the book. Students in 2022 used an early draft of this book, and Jae Kim, Alana Mittleman, and Zhuoneng Wang found some small errors that were fixed. Two project groups in 2022, consisting of Jae Kim, Jason Hsu, Tianying Lyu, Simon Corpuz, Auli Shen, Sarah Chen, Jeni Huang, and Junqi Wang, directly wrote early versions of two chapters, so I want to thank them for letting me use their material. The teaching assistants for the class had a big influence on the assignments and the content: Jeffrey Rzeszotarski, Kathy Yu, Asit Parida, and Chaoran Chen. I would especially like to thank Chaoran Chen for creating ScrollTest and letting me use parts of his paper about it. I would also like to thank all the guest speakers for that class, most of whom allowed me to use their content for the interview chapters in this book: Bill Atkinson, Loren Brichter, Dan Bricklin, Bill Buxton, Norm Cox, Rob Haitani, Chris Harrison, Don Hopkins, Bruce Horn, Julia Schwarz, Ted Selker, David Canfield Smith, Larry Tesler, Tandy Trower, Gregg Vanderheiden, and Shumin Zhai.

I carried out all the work on this book while a faculty member in the Human–Computer Interaction Institute (HCII) at Carnegie Mellon University (CMU), and I would like to thank the many department heads of the HCII who gave me time off from teaching over the years, as well as permission to invent and teach an elective course like *Interaction Techniques*: Dan Siewiorek, Justine Cassell, Anind Dey, and Jodi Forlizzi.

The content of this book was inspired by two of my important early works. First was the "All the Widgets" video I made for the ACM SIGCHI 1990 conference [Myers 1990b], which was supported by the conference organizers and CMU's audiovisual team, and had contributions from many people in the community. Second was the 1998 paper "A brief history of human computer interaction technology," from *ACM Interactions* [Myers 1998a], which was adapted for Section 2.5. The original paper benefited from feedback from many people including Stacey Ashlund, Meera M. Blattner, Keith Butler, Stuart K. Card, Bill Curtis, David E. Damouth, Dan Diaper, Dick Duda, Tim T.K. Dudley, Steven Feiner, Harry Forsdick, Bjorn Freeman-Benson, John Gould, Wayne Gray, Mark Green, Fred Hansen, Bill Hefley, D. Austin Henderson, Jim Hollan, Jean-Marie Hullot, Rob Jacob, Bonnie John, Sandy Kobayashi, T.K. Landauer, John Leggett, Roger Lighty, Marilyn Mantei, Jim Miller, William Newman, Jakob Nielsen, Don Norman, Dan Olsen, Ramesh Patil, Gary Perlman, Dick Pew, Ken Pier, Jim Rhyne, Ben Shneiderman, John Sibert, David C. Smith, Elliot Soloway, Richard Stallman, Ivan Sutherland, Dan Swinehart, John Thomas, Alex Waibel, Marceli Wein, Mark Weiser, Alan Wexelblat, and Terry Winograd. Editorial comments on that article were also provided by the foregoing as well as by Jim Hollan, Ellen Borison, Rich McDaniel, Rob Miller, Bernita Myers, and Yoshihiro Tsujino.

Finally, I had many questions while writing this book and fortunately knew many experts in the field who were able to answer them: Tom Anderson, Bill Atkinson, Rob Baecker, Michel Beaudouin-Lafon, Eric Bier, Nathaniel Borenstein, Bill Buxton, John Canny, Stuart Card, Michael Coblenz, Norm Cox, Don Hopkins, Matt Huenerfauth, Hiroshi Ishii, Jeff Johnson, Ted Kaehler, Robert Kraut, Nikolas Martelaro, Jim Morris, Tom Neuendorffer, Chris Neuwirth, Jakob Nielsen, Don Norman, Chris Schmandt, Ben Shneiderman, David Canfield Smith, and Gregg C Vanderheiden. The final version was much improved by the careful editing at ACM Books—thanks especially to Sean Pidgeon, Brent Beckley, Karen Grace, Bernadette Shade, and Nicole Valenzi.

Systems and websites I used to create this book include Microsoft Word for Windows (various versions over the years, including Professional Plus 2013 and 2016, and then Word 365 Apps for Enterprise), which I am grateful was able to handle a single file containing the entire book of over 600 pages with over 550 embedded pictures, with its support for tables of contents and figures, indexing, cross-references, incremental search, and global search and replace throughout the whole book. The resulting file was over 275MB on the disk. I also used Endnote to handle my bibliography, which similarly was able to scale up to a book-size project.

I made extensive use of Wikipedia, but mostly as a way to find out dates and primary sources. As Wikipedia entries can be changed at any time, I much prefer to verify and reference the original citations, which is what was used in this book. Still, Wikipedia, along with the "Wayback Machine" or "Internet Archive," archive.org, are invaluable resources, to which I recommend everyone who can should make financial contributions so they can stay available and advertisement free.

Preface

Overview:

This book provides a comprehensive study of the many ways to interact with computers and computerized devices. The opening chapter defines and motivates these *interaction techniques* and why they are important. Part I then provides a comprehensive but brief history of desktop and mobile interaction techniques and the important historical systems and interaction styles that hosted them. Part II looks in detail at the various kinds of interaction techniques, with each chapter focusing on a different one. The chapters explain the history and design variations and considerations, as well as the constraints that drive the designs: pointing devices, scrolling, percent-done progress indicators, menus, text entry, forms and dialog boxes, graphical editing, window managers, three-dimensional interactions, ubiquitous computing, gesture recognition, undo, help, assistive techniques for people with disabilities, and artificial intelligence techniques. Part III focuses on using, designing, and implementing interaction techniques, including more "meta" issues about interaction techniques in general. Much of this part is focused on programmers who will need to write code to use interaction techniques or to make new ones work. Interaction designers and graphic designers, even if they are not programmers, might need to know some of the material in this part to help evaluate existing interaction techniques or design and evaluate new ones. Finally, Part IV summarizes interviews with the original inventors or designers of some interaction techniques, based on guest lectures they gave to my class. As these are based on those lectures, I was only able to include people who agreed to talk and also agreed to have their lecture recorded (which unfortunately eliminated May-Li Khoe's talk about the Apple Force Touch).

All of the content of this book has been carefully researched, and full citations to original sources are included throughout. As mentioned in the Acknowledgments, in many cases the information comes from the original people who worked on the interaction techniques.

Who is this book for:

For historians of technology and journalists: This book will provide value to anyone interested in the user interfaces (UIs) of computers and other technology. It was designed to capture the *history* of interaction techniques, which somewhat surprisingly has not been previously captured. For example, even though there are many articles about the early Smalltalk and Xerox Star systems and their many influences, I have never seen a detailed discussion of how their interaction techniques worked. The extensive citations to original sources can help those wanting even more detail than is provided here.

For designers of UIs: Virtually all UIs are made out of interaction techniques, so everyone designing or implementing a UI of any kind will be using the interaction techniques discussed in this book. Also, as most UI design is creating or using interaction techniques, the topics discussed in this book are relevant to UIs in general and not exclusively for interaction techniques. Part III provides guidance about how to use and implement UIs with interaction techniques.

For inventors of new interaction techniques: New interaction techniques are being invented all the time, and this book motivates where and why new ones are needed. It presents the constraints and tradeoffs relevant to new designs in many different areas and the industry-standard ways to evaluate the quality of the interaction techniques. New techniques can have a high economic value, and this book also includes a discussion about legal issues around interaction techniques such as patents and copyrights.

For lawyers working on intellectual property cases about UIs: There are many patents and lawsuits about interaction techniques, and patent attorneys may benefit from understanding the history and details of the historical and current interaction techniques through time.

For students and faculty of UI courses: This is a unique reference that brings together detailed discussions about designs and tradeoffs for UI elements. It may be the primary reference for classes about interaction techniques or a supplementary reference for other classes where an understanding of their low-level designs and/or implementations would be useful. As mentioned below, full teaching materials on these topics are available.

For users of interaction techniques (which is everyone using technology): Few people are aware of the details about how the interaction techniques on their technology works. Throughout the discussions of each interaction technique, this book provides lots of neat tricks that make interacting with devices faster, less error prone, and more efficient. For example, many people do not know how to undo erroneous actions on an iPhone (by shaking it), how to abort an action while it

is in progress on Microsoft Windows (by hitting the keyboard Esc key) or how to select a range or multiple individual items in a list (with Shift-Click or Ctrl-Click). For users with disabilities, Chapter 17 focuses on ways to adapt interaction techniques for specific abilities. All of these and many more details are explained for each interaction technique.

Use in classes:

This book comes out of my class on *Interaction Techniques*, which I have taught four times (so far). The most recent class used this book as the primary reference for all the classes. The full curriculum for my class is available at: https://www.cs.cmu.edu/~bam/uicourse/05440inter/.

Included there are the full curriculum, lecture slides, and homeworks for each year. Others have used my material for their own courses, which is explicitly authorized (with attribution). For example, Associate Professor Shiri Azenkot twice used my material for her course on "Future Interaction Techniques" at Cornell Tech.

As stated in my course curriculum, after reading this book students will be able to:

- Describe the historical progression of the most important interaction techniques and the factors that impacted their evolution and eventual widespread adoption.
- Articulate design issues regarding interaction techniques.
- Design a new interaction technique given a set of requirements and constraints.
- Evaluate interaction techniques using the appropriate tests for performance and usability.

CHAPTER 1

Introduction

1.1 Importance of the Topic

Consider the lowly checkbox in Figure 1.1. How many states can it be in? An obvious answer is two: for checked and not checked. But the correct answer is more like 48 different states! As shown in Figure 1.2, a checkbox can show it is *neither* checked nor not checked—for example, if text is selected that is *partially* crossed out. So that is three states instead of two. But it turns out that checkboxes show other things as well. For example, if the mouse is placed over a checkbox, then the outline turns blue to show the mouse is *hovering* and something will happen if you click (Figure 1.3(a)). This blue glow happens no matter the checked status, so that means there are 2 (hover and not) times 3 (check states) = 6 possibilities so far. But if you *press down* the mouse button over the checkbox and hold it, then the box is filled with blue to show that an interaction is in progress (Figure 1.3(b)) and there is also a *keyboard focus* (Figure 1.3(c)) because checkboxes in Windows can be controlled entirely using the keyboard without using the mouse (see Section 7.3.11.4). Finally, checkboxes can be *grayed out* when they are inactive and cannot be selected. Again, these are mostly independent of the checked status and from each other, making about 2^4 * 3 = 48 possible states! This book discusses how that amount

Figure 1.1 A checkbox that is "checked." Source: Screenshots from PowerPoint 2016 on Windows 10 on a PC. Used with permission from Microsoft.

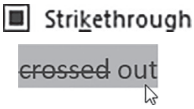


Figure 1.2 This checkbox shows a square to indicate it is neither checked nor unchecked since both strikethrough and not strikethrough text are selected. Used with permission from Microsoft.

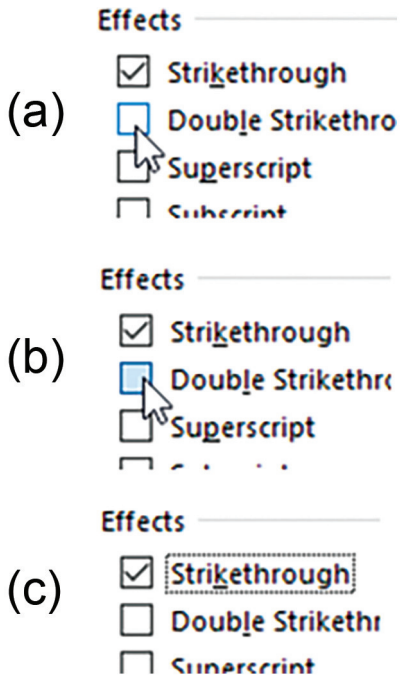


Figure 1.3 Checkbox showing the (a) blue mouse hover, (b) mouse pressed down, and (c) text focus outline. Used with permission from Microsoft.

of complexity came to be and what designers must consider when creating a new kind of checkbox or even when putting a checkbox into their user interfaces.

Next, consider the *copy-and-paste* interaction technique. Even though this is used billions of times by virtually all computer users, when you think about it this is somewhat odd. The *copy* part of the operation takes what is selected and puts a copy into an invisible *clipboard*, which is nothing like what happens when you use a physical photocopier, for example, which takes a piece of paper and gives you another piece of paper you can see.[1] And consider *paste*, which means to stick something down with glue, like we did in kindergarten. How did this interaction come to be? This is the story of Larry Tesler, who invented the interaction way back in 1973, as told in Chapter 23. But that is "old news"; what about *new* interaction techniques—in fact, new ones are being invented all the time! For example, Loren Brichter invented *pull-to-refresh* in 2010, and this has now almost completely replaced the formerly ubiquitous refresh icon on smartphones (Figure 1.4), as told

1. The operation of taking something and making another one that is identical must be called *duplicate* on computers and should not be called *copy*.

Figure 1.4 This refresh icon was formerly used frequently on smartphones—for example, to check for new mail or new Facebook messages—but has been mostly replaced by "pull-to-refresh," discussed in Chapter 32. Source: screen capture.

in Chapter 32 (see also US Patent 8,448,084). Chapter 12 discusses interaction techniques being invented today for the new modalities of artificial and virtual reality, and even for today's modalities, there are new interaction techniques being introduced with almost every new version of their operating systems.

Now consider *touching* a smartphone screen. What will happen? Obviously, this question cannot be answered in general since it depends on what is on the screen under the finger. Is it a button or a bit of text or an empty field? But on many (but not all!) phones, it also depends on how *hard* and how *long* you press—a quick *tap* performs an action different from a *long-press* and may also be different from a hard *3D* press on previous versions of the iPhone. Not to mention that if the touch is the start of a *swipe* movement across the screen or part of two fingers touching, then something entirely different than any of these will happen.

This book tells the story of all the *interaction techniques* that make up how we use all electronic devices, from regular computers to "dumb" and smart phones to consumer electronic devices. It explains why these are the way that they are, which will be of interest to anyone who cares about the history of technology. It also explains how interaction techniques operate now and will in the future, which will be of interest to people who use devices (that is, almost everyone) and to people who *design* new systems and need to know the issues and constraints that previous designers considered.

This book also covers how to *evaluate* interaction techniques (see especially Sections 4.14, 8.5, and 21.4) so readers will know when their devices could be doing things in a better way. This will also help designers know if their new designs are better than existing approaches.

I think it is interesting that even though there are many, many books and articles about the history of computers, their user interfaces, and the people involved (e.g., Goldberg [1988], Press [1993], Myers [1998a], and Isaacson [2015]), none have included how the interaction techniques were created and evolved. This book fills in that gap.

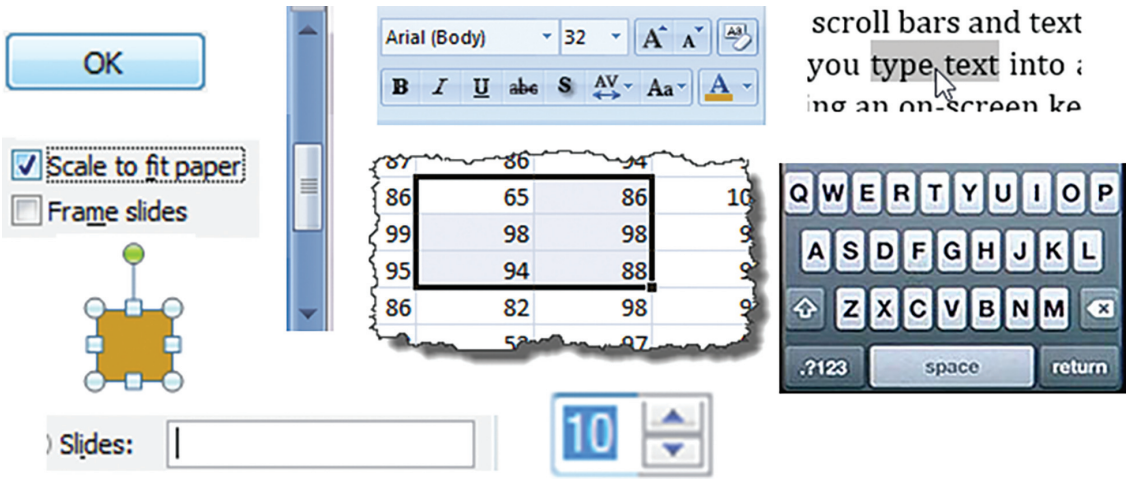


Figure 1.5 Some examples of input techniques on computer or smartphone screens. Sources: Screen captures.

Figure 1.6 Some examples of physical interaction techniques for consumer electronics and games. Source: Original composite.

1.2 What Are Interaction Techniques?

But what specifically is an *interaction technique*, and how is it different from a regular *user interface*?

Informally, an interaction technique is an object on a computer screen that you use to provide input to the computer. Typical examples include buttons like the *OK* button in Figure 1.5, checkboxes, scrollbars, and text input fields. But *interaction techniques* also encompass how you type text into a regular computer (with a keyboard) or into a mobile phone via an on-screen virtual keyboard or the *Swype* gesture technique (described in Chapter 8). Using copy, cut, and paste to edit text or drag-and-drop to move something are also interaction techniques.

In addition, when you swipe on a smartphone to scroll up or down, or swipe left or right to show you like or do not like something, those are interaction techniques. So is the way you draw, select, and edit graphics on the screen in computer applications like Microsoft PowerPoint or Adobe Illustrator. So are the physical controls like buttons on a remote control, a slider on a dimmer switch, and joysticks on a game controller (see Figure 1.6).

1.3 Formal Definitions

With that in mind, how would we define *interaction technique* more formally? First, we need to define a few other useful terms.

1.3.1 User and Designer

A *user* is a person who will use a product, whereas a *designer* is the person who creates (or designs) the product. (There might also be an *implementer* separate from the designer who actually implements the design, but this book uses "designer" for both roles except in Chapters 20 and 21, which focus on implementation.) Of course, a designer will be a user of various tools—for example, a programmer creating a new scrollbar interaction technique will use text entry interaction techniques to type and edit the program code. In the more typical case, the user will be a consumer using a smartphone to send an email or play a game, in contrast to the designer who might be a highly trained programmer.

Note that we are using the broad term *product* rather than just a *computer*—we intend to include the use of anything on a computer or smartphone (all applications and apps), any consumer electronics device or appliance, and even the use of non-technology things. For example, it is appropriate to consider the "user" of a spoon or chair, which becomes a topic relevant to this book when those devices need to be adapted (often with technology) for people with disabilities (discussed in Chapter 17). We also will use the terms *consumer electronics* and

appliance interchangeably, even though appliances do not necessarily need to be electronic since they will typically all use the same kinds of interaction techniques (like knobs, switches, and displays).

1.3.2 User Interface and Usability

A *user interface*, or UI, is everything the user encounters when using a product. This includes all of the following aspects (among others):

- Functionality and usefulness
- Content
- Labels
- Aesthetics
- Presentation
- Layout
- Navigation
- Speed of response
- Emotional impact
- Documentation & Help

This book will often refer to the *quality* of a UI, which is often called the *usability* or *user experience* (UX). Usability attempts to measure the many properties of a UI, including:

- *Learnability*—how quickly a user can learn to use it.
- *Efficiency*—after learning, how quickly it can be used by experts.
- *Productivity*—how much can be done with it per unit time.
- *Memorability*—if it is used and then not used for a while, when the user goes back to it, how easy it is to remember how to use it.
- *Errors*—how likely it is that the user will make an error when using it.
- *Satisfaction*—how much do users like using it.

Often, learnability will include how apparent it is to a novice user what to do. Don Norman [1988] has called this property the *affordance* of the presentation. For an interaction technique, often the affordances help users know how it will behave. Most interaction techniques have aspects designed to be learnable by novices and other aspects designed to help experts be more efficient—for example, most menus have textual labels that novices can read to find out what the options are, along

with keyboard accelerators (like *control-s* for "Save") so experts can enter the commands quickly (see also Chapter 21 for a discussion of methods for designing and evaluating these properties).

1.3.3 Pointing Device

A *pointing device* allows a user to *indicate positions on the screen*. Examples include a touchscreen on a smartphone or a mouse used with a regular computer. Chapter 4 discusses pointing devices in depth.

1.3.4 Input and Output

It is worth mentioning that the words *input* and *output* are always used from the computer's point of view (even though we focus on users). Thus, *input* is information *into* the computer typically *from* the user, and *output* is information that comes out *from* the computer and goes *to* the user. (These terms can also be used for information to and from the network or another computer, but that is less relevant to this book's topic.)

1.3.5 Graphical User Interface

A *graphical user interface*, or *GUI*, is a *UI that uses a pointing device and graphical objects*, where *graphical objects* are basically all the things you see on the screen (including text). GUIs include the use of all the screen-based interaction techniques.

1.3.6 Human–Computer Interaction

Human–computer interaction (HCI) is the study of the design and use of technology, especially the interface between people and devices, and encompasses the study of interaction techniques. Note that this definition is broader than just "computers" since I am including the study of all technology. HCI is sometimes also abbreviated *CHI*, pronounced "kai." An older term that is no longer used, for obvious reasons, is *man–machine interfaces* (MMI). Related fields include *human factors* and *ergonomics*, which historically have had more focus on the physical aspects of interaction [Grudin 2017]. For a more complete overview of HCI and related disciplines, see various textbooks such as Nielsen [1993], Hartson and Pyla [2012], Grudin [2017], and Shneiderman et al. [2017].

The main conference in the HCI area is the *ACM SIGCHI Conference on Human Factors in Computing Systems* or just *CHI* for short (see, e.g., chi2022.acm.org/), sponsored by the *Association for Computing Machinery* (ACM). ACM is the largest association for professionals and researchers in computing, and SIGCHI is a

subgroup of ACM called the *Special Interest Group on Computer–Human Interaction*. The annual CHI conference dates back to the 1980s and is now attended by about 2,000 to 3,000 people every year, and I mention it because it is one of the chief places that research about interaction techniques is published. The other is the smaller *ACM User Interface Software and Technology Symposium* or *UIST* (pronounced "wist"), which focuses on the more technical aspects of HCI (see www.uist.org), but papers about interaction techniques might also appear in more specialized conferences.

1.3.7 States and State Diagrams

In the introduction, I discussed the number of *states* that a checkbox can be in. Here, I am using the word *state* to mean a *stable configuration of a UI or input device*. The system can *transition* between states as a result of user or system *events*. The set of valid states and transitions for a system can be represented using a *finite-state machine* (FSM), also called a *state machine*, *state diagram*, or *state transition diagram*. Although there is significant theoretical modeling around state machines, and many variants that can express different kinds of behaviors, we will just use relatively simple versions fairly informally to try to make the behaviors clearer. There is a more complete discussion in Section 19.4.

Figure 1.7 shows a simple state diagram for a one button mouse. It has two states—the start state, where the button is not down, and a button-down state. The user transitions between these states by pressing or releasing the mouse button. The diagram also shows that the mouse can move in either state, but this does not cause a state change. Some state diagrams also annotate the states and/or transitions with the *actions* that happen.

1.3.8 Interaction Technique

Now to the main event!

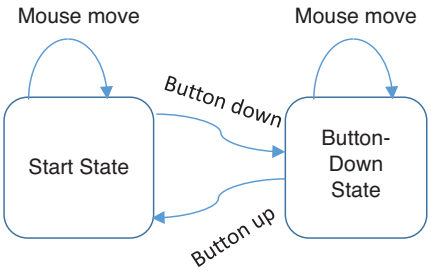


Figure 1.7 Simple state diagram for a one-button mouse. Source: Original.

My definition of interaction technique:

> An *interaction technique* starts when the user performs an action that causes a product to respond and includes the direct feedback from the product to the user. Interaction techniques are generally reusable across various applications.

Wikipedia's definition: (as of June 13, 2022)[2]

> An **interaction technique**, **user interface technique** or **input technique** is a combination of hardware and software elements that provides a way for computer users to accomplish a single task.

My problem with Wikipedia's definition is that it is too vague and general, especially with respect to what it means to be a "single task." Also, I never really heard them called a "user interface technique."

Although it is tempting to abbreviate interaction techniques as "IT," that acronym has a well-defined meaning in computer science (as "information technology"). We therefore adopt the abbreviation "**IxT**," considering that "IxD" is a well-accepted abbreviation for "Interaction Design" (see, e.g., its Wikipedia entry).

1.3.9 Other Names

Over the years, a variety of other names have been used for, at least partially, what we are calling *interaction techniques*. For example, I called them *widgets* in my 1990 video "All the Widgets" [Myers 1990b]. Today, the word *widget* is sometimes used for small applications, sometimes called *dashboard widgets*, like *desk accessories* that might be put on the desktop of a computer, like a calendar or clock app, which would not be classified as an interaction technique. However, we will continue to use the word *widget* in this book to refer to *small interaction techniques that have a concrete graphical presentation*. Thus, an on-screen checkbox (Figure 1.1) is a widget, but the way to do copy-and-paste in a text editor is an interaction technique and is not a widget.

Another popular term is *control*, which was popularized mostly by Microsoft Windows. For example, the list of items one can drag into a UI using the Visual Studio editor is called a list of controls (see Figure 1.8). Some of what Windows calls *controls* seem too simple to me to be classified as an interaction technique or widget, like the pointer or groupbox decorations, and others seem too complex,

2. https://en.wikipedia.org/wiki/Interaction_technique. A key problem with references to Wikipedia is that entries can change at any time, so anything there may be different from when I looked at it. That is one reason this book rarely relies on Wikipedia and tries to only use original sources that won't change.

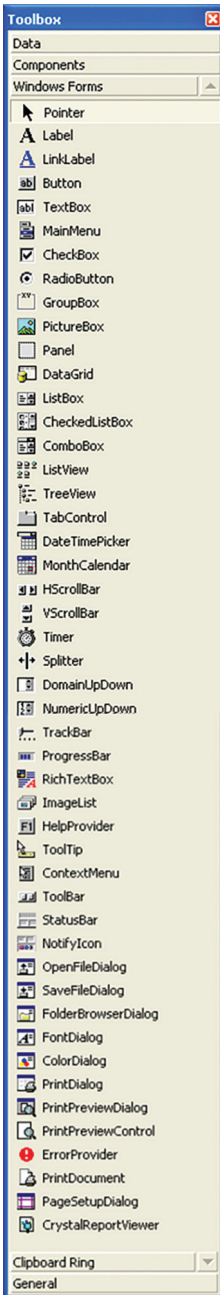


Figure 1.8 Some controls in the menu for Microsoft Visual Studio. Source: Screen capture. Used with permission from Microsoft.

like a full PageSetupDialog. Wikipedia seems to sometimes use the term *graphical control element* or *graphical widget* for what I am calling *widgets*.

I have also heard these called *gadgets*, which again most people think are something else. Other terms that seem to me to be too generic include GUI *elements* or *tools*. *Tools* might come from the use of the word *toolbox* or *toolkit* to describe the software that provides a collection of interaction techniques, as discussed in Section 20.4.

Other terms I do not like include *components* and *behaviors*. *Components* implies to me larger, heavy-weight parts of an application such as the graphing component of Microsoft Excel, which includes a large number of interaction techniques. I will use *behaviors* to refer to the way that interaction techniques respond (the way that they *behave*), which may be independent from the way that they *look*. Thus, behaviors are the *feel* part of *look and feel*.

Recently, the term *micro-interaction* (sometimes written as *microinteraction*) has started to be used for small changes to the UI, often through animations or visual change, that trigger on user interactions and provide clear feedback [Saffer 2013]. An example would be when a toggle switch slides and changes color when turning on a setting.

1.4 What Is *Not* an Interaction Technique?

So the term *interaction technique* is quite broad—what is *not* an interaction technique? The term does *not* cover whole applications, like Microsoft Word, that allows you to edit a document. Obviously, Word uses *a lot* of interaction techniques, but the whole program does not count as an interaction technique.

As mentioned above, neither is a whole component of an application, like the charting part of Microsoft Excel.

I am generally not including *visualizations* as interaction techniques. A *visualization* is *a visual presentation of information intended to make it easier to understand*. I do not include these since they have no *interaction*—they are just the presentation. On the contrary, interactive visualizations, where the user can manipulate the view or the underlying data, would contain interaction techniques, but these are mostly conventional interactions that are already covered in other chapters. There are many good summaries of visualizations—for example, Myers [1986a], Card et al. [1999], Bederson and Shneiderman [2003], and Bonneau et al. [2006].

1.5 Why Am I Writing this Book?

I have always found interaction techniques interesting. I personally have used, studied, and even invented interaction techniques dating back to 1975. Having taught a course on *Interaction Techniques* four times (as of 2022), it seems useful to capture the organized knowledge about them in a book.

When I was an undergraduate student at the Massachusetts Institute of Technology (MIT) between 1975 and 1979, I was a student undergrad researcher (as part of the Undergraduate Research Opportunities Program, UROP) at what was then called the Architecture Machine Group (ArchMac), and is now the MIT Media Lab. One of the projects I worked on was the Spatial Data Management System (SDMS) [Herot et al. 1980], which was the first system to have a general-purpose *zoomable user interface* (ZUI) (see Section 11.11.1.2).

Later, I was selected to work at the Xerox Palo Alto Research Center (PARC) as part of an MIT co-op program. Xerox PARC is a really important research lab where many aspects of UIs and hardware were invented and refined, including window managers, laser printing, integrated development environments, icons, copy-and-paste, Ethernet, ubiquitous computing, and many others (see Chapters 2 and 3). I was fortunate to be there while other people were doing these important things. In particular, I worked in the Smalltalk group [Tesler 1981] during the summer of 1977 and the Mesa group in 1978–1980, and I used the Alto computer and its mouse (see Figure 1.9). I also saw a demonstration of the Xerox Star system before it was released (see Section 2.4.6).

After I graduated from MIT, my next stop was at a startup computer company in Pittsburgh called Three Rivers Computer Corporation, which was a spin-off from Carnegie Mellon University (CMU). I was there from 1980–1983, and my primary contribution relevant to this book was to create one of the first commercial window managers called Sapphire [Myers 1984, 1986b]. Innovations in Sapphire included the ability to control all aspects of the windows just with the keyboard, a novel

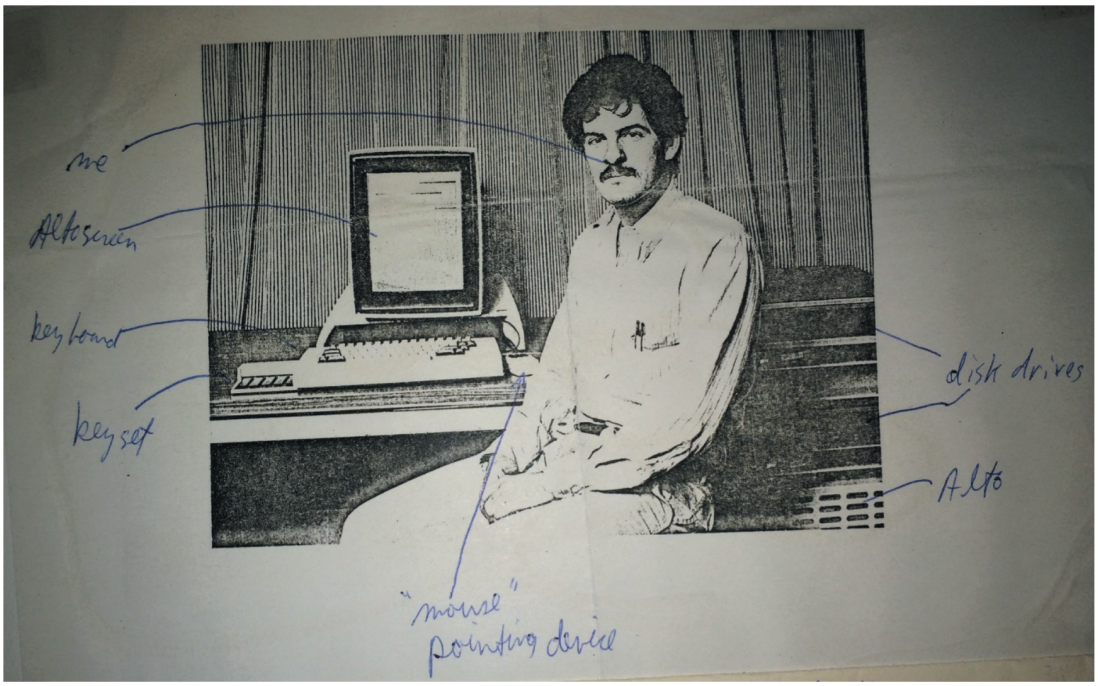


Figure 1.9 Picture of me in front of the Alto computer at Xerox PARC in 1979. I sent this annotated picture to my father (in a physical mail letter!) since people were not familiar with the technologies back then. See also Chapter 2. Source: Original photo and annotations by the author.

design for the icons, progress bars in the icons and window titles, and collecting the icons together in a window (see Chapter 11). Many of these features were subsequently used by many windowing systems. I later wrote a paper that helped explain and define the terminology of windows and their interaction techniques [Myers 1988].

In 1983, I went back to school to get a Ph.D. at the University of Toronto. One of my first publications was about *percent-done progress indicators* or *progress bars* (see Chapter 6), which was highly influential [Myers 1985b]. I also collaborated with Bill Buxton on a highly cited paper about using *both* hands simultaneously for interaction techniques, including scrolling, selecting, and resizing [Buxton and Myers 1986]. My Ph.D. dissertation was about new ways to make it easier to design and implement novel and conventional interaction techniques [Myers 1987a, 1987b, 1990c] (see Section 20.8).

After getting my Ph.D., I moved to CMU. My first projects included tools to help programmers implement interaction techniques and other UI software [Myers 1989, Myers et al. 1990, 1997a] (see Chapter 20). My students and I have also created a variety of new interaction techniques over the years (e.g., Wobbrock et al. [2002, 2003, 2004], Ko and Myers [2006], Gonzalez et al. [2007], and Lockerman et al. [2017]).

Another notable activity was the video I made in 1990 called "All the Widgets" [Myers 1990b], see also https://youtu.be/9qtd8Hc90Hw or https://vimeo.com/61556918. As mentioned above, the annual SIGCHI conference is the main forum for HCI work, and I was appointed to be the Video Chair for CHI'1990. At the time, there was no World Wide Web or DVDs, so all videos had to be distributed on physical videotapes. In addition to the paper proceedings for papers, the conference published a video proceedings of videos that went with papers and also separate stand-alone video publications, most of which were between 3min to 15min long. These were published as part of the ACM SIGGRAPH Video Review.[3] I proposed to create a special video (in addition to the regular conference videos) about the history of various interaction techniques such as scrollbars, buttons, and window managers. The SIGCHI organization agreed to pay to produce it, and I used my connections to invite all the major players to contribute snippets about their innovations. The resulting 2-hour, 15-min video contains over 175 segments from 30 systems from 16 companies and provides a comprehensive record about how the various interaction techniques had developed up to that point [Myers 1990b]. Surprisingly, this is the only record for how even some well-known interaction techniques work. For example, in spite of the significant amount published about the Smalltalk environment (e.g., Tesler [1981]), I have not been able to find any other record about how the Smalltalk scrollbars looked and worked.

Other important activities relevant to interaction techniques include a paper in 1998 summarizing the history of HCI technology [Myers 1998a], which formed the basis for Chapter 2, and my semester-long course on *Interaction Techniques*, which I have offered four times, in 2014, 2016, 2019, and 2022. As part of the course, I invite luminaries and inventors of interaction techniques to give guest lectures, and many of the interviews in this book come from those. The complete syllabus and videos of the guest lectures are available from the course website:

- http://www.cs.cmu.edu/~bam/uicourse/05440inter/

3. SIGGRAPH is the ACM **S**pecial **I**nterest **G**roup on **GRAPH**ics. SIGCHI made use of SIGGRAPH's pre-existing facilities for publishing and distributing the videos.

Chapter 20 is about implementation techniques and tools, which is also an area in which I personally have done much research and teaching. I created a number of early influential UI tools, such as Peridot [Myers 1990c], Garnet [Myers et al. 1990], and Amulet [Myers et al. 1997a], and have regularly taught courses on *Advanced User Interface Software* and *Software Structures for User Interfaces*. See:

- https://www.cs.cmu.edu/~bam/#courses

Finally, Chapter 22 is about patents and other intellectual property. I personally am listed as an inventor on three patents, at least one of which (US Patent No. 7,729,542) is clearly a new interaction technique (as a new way to enter text for people with disabilities). I have also been an *expert witness* in legal matters about patents on UI topics, including interaction techniques. As of 2022, I have been an expert witness on over 80 intellectual property cases, which include giving testimony at two jury trials and five other hearings in front of judges.

1.6 Value of Interaction Techniques

In addition to the reasons discussed in Section 1.1, there are other reasons that interaction techniques are important. One stems from their ubiquity—a tiny change in an interaction technique can have an enormous impact on billions of users, and thus a huge impact on people's productivity. For example, if a new design saves a few milliseconds in an operation that every smartphone user does all the time, like scrolling a list or deleting an email message, this will have a huge multiplicative effect.

Furthermore, interaction techniques can have an enormous *monetary value* for companies. Chapter 22 discusses *intellectual property* such as *patents*, which are a way to protect how an interaction technique works to prevent others from copying it. For example, one of the important patents that Apple owns is on how scrolling works on the iPhone (see Section 5.9), where there is a *bounce* at the end of the list to show that one is at the end (US Patent number 7,469,381 and others). This was one of the patents that Apple asserted on Samsung in some lawsuits, and Apple was awarded $1.05 billion in damages in one case for five patents in 2012 [Guglielmo 2012] and $290 million in another case on different patents in 2013 [Williams 2013]. These numbers were adjusted on appeal, but no matter the exact amount, this is a lot of money for interaction techniques!

1.7 Need for New Interaction Techniques

I argue that the world really needs some new interaction techniques, and hopefully this book can help motivate research in that direction.

For example, the *desktop metaphor* (see Chapters 2 and 11), where users invoke apps by clicking on icons, was invented about 40 years ago and is still in use on today's smartphones even though the separation of the data of each app into separate silos causes significant inefficiencies for users, requiring copy-and-paste or re-entry of data across multiple apps for many tasks. What would be a better way to organize a set of functionality that users could access? Chapters 18 and 39 have some proposals.

Another example of significant need is text entry for mobile devices. The standard QWERTY keyboard was invented in 1868 and is still used on smartphones even though it is very inefficient (see Chapter 8). How might textual content be better entered into smartphones (and regular desktop computers for that matter)? Speech recognition is an easy proposal (Chapter 18), but it has limitations and people are not particularly good at dictating coherent sentences. What else might be used?

Many other limitations and inefficiencies of today's interaction techniques are discussed throughout this book, some of which are under active research, published at the major HCI conferences (see Section 1.3.6). What are the next new ideas?

1.8 Why Are Interaction Techniques Hard to Design?

1.8.1 Many Needs

It might seem like it would not be that hard to design an interaction technique, but there are many considerations. First is the wide variety of different needs that each interaction technique must support. This was demonstrated by the discussion of the checkbox (Section 1.1), which not only needs to show checked or not but also needs to support showing the user the various intermediate states (like hover and pressed-down) and multiple ways of interacting (e.g., with a mouse or a keyboard). In fact, it is not uncommon to find implementations of menus and checkboxes that forget to handle some situations, such as pressing down inside and then moving outside and releasing (try this on your favorite custom-created web menus—they often get confused). This is an example of the large number of *design decisions* that an interaction technique designer must consider.

1.8.2 Details of the Behavior

Further, each design decision must be carefully made since it may have a significant impact on the user's performance. For example, how far should a list scroll when the user flicks it on a smartphone? The answer for a well-designed device

depends on many factors including the length and speed of the flick and the size of the list. Developing this algorithm requires significant experimentation and testing so that it works for scrolling to the next item as well as to the 1,000th item. This feels very natural even on the original iPhone, but I remember using an early Nokia phone that was released just after the iPhone where scrolling felt like the items were stuck in mud since the scrolling was sluggish and non-responsive, showing that the designers had not spent the necessary time and effort on testing and refining it. Another example is the eight years of research it took IBM to make the pointing stick work smoothly (see Section 4.8 and Chapter 34).

1.8.3 Abort, Cancel, and Undo

Interaction techniques also operate over *time*, so the behavior must take into account the collection of things that users do. For example, consider again the checkbox—users might press down and hold a mouse button over it, and then there are many different things the user can do next, which the interaction technique designer must consider, such as moving the mouse away while still holding down, and releasing the mouse button back over the checkbox or over somewhere else entirely.

The designer must decide whether the user has the ability to *abort or cancel* the interaction technique while it is in progress, which means to stop the interaction and not do anything. For the checkbox, if the user moves outside of the checkbox and then releases the mouse button outside, this (usually!) aborts the behavior, and the checkbox does not change its value. More complex interactions like dialog boxes (Chapter 9) typically will have a Cancel button. Aborting is covered in more detail in Section 15.4.

A related design option is whether the interaction can be *undone*, which means to reverse the operation of the interaction after it has been completed. For the checkbox of Figure 1.3, the operation of causing the text to be shown with *Strikethrough* (~~like this~~) can be undone, which will cause the text to revert to not being Strikethrough, and will also undo the checking of the checkbox. Other interaction techniques, like scrolling, are typically *not* able to be undone, except by the user manually trying to go back to where they were. Chapter 15 discusses Undo in depth.

UI design professionals have always recommended that Cancel and Undo be supported, and this requirement also appears in most guidelines (e.g., Nielsen and Molich [1990]). However, support for Cancel and Undo have mostly *not* been provided in most web and smartphone interactions [Norman and Tognazzini 2015]. How could they be put back in?

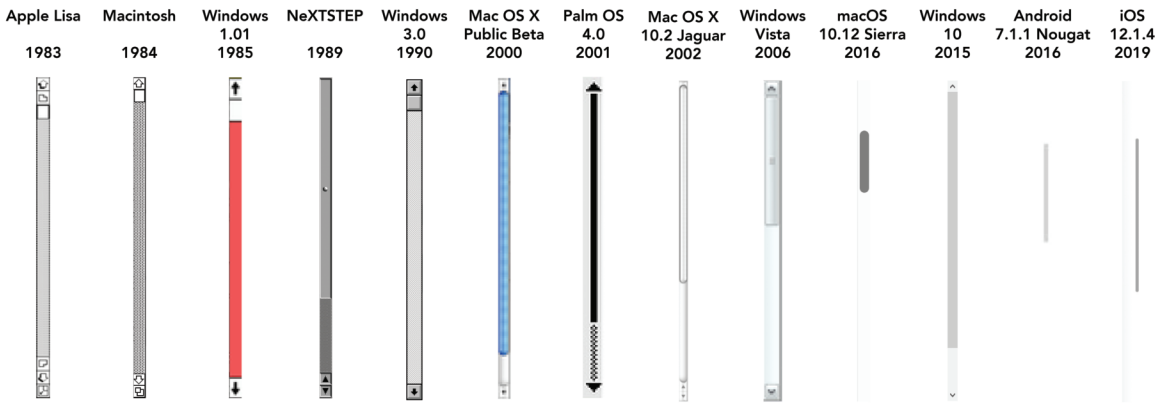


Figure 1.10 Many looks for scrollbars over the years. Source: Original from screenshots and open source.

1.8.4 The Look

In addition to these design decisions about the *behavior* (the *feel*) of the interaction techniques, the designer must also consider how it *looks*. This includes the graphical presentation of what is shown on the screen, both before the user operates the interaction technique and the feedback while it is being operated. As shown in Figure 1.10, there can be many different ways that scrollbars can look, which must be designed (see also Chapter 5). Some of the considerations for the look include what icons to use to signify actions (e.g., whether the arrows at the ends face up or down), where the various elements should be placed (e.g., the NeXTSTEP system [Section 2.4.11] put both arrows at the bottom), the use of colors, and the overall graphical style, which has varied from *clean lines* in the Xerox Star System (Section 2.4.6), through a more three-dimensional (3D) look starting with NeXTSTEP (called *skeuomorphism*, or making items look like their real-world counterparts), to the *flat design* of Macintosh OS X Lion.

The look is an important part of the *affordance* of the interaction technique. As mentioned above (Section 1.3.2), the affordance is the *perceived and actual properties of the thing, primarily those fundamental properties that determine how the thing could possibly be used* [Norman 1988, p. 9]. When the look has good affordances, then *the user knows what to do just by looking*.

Another dimension of the look is the support for *animation* in interaction techniques, which means the way that the graphics change over time. For example, after the user selected a menu item on the original Macintosh, the item blinked before the menu disappeared. Similarly, today, menus often are animated by fading in and out (see also Chapter 7).

1.8.5 Localization and Internationalization

Most UIs are required to work across multiple natural languages, like English, French, Chinese, Arabic, and so on, which will affect all interaction techniques that contain or operate on text (like menus, buttons, checkboxes, text editing, etc.). A key issue is that text with the same meaning in different languages will have different physical sizes on the screen, and may need to be oriented differently (e.g., Arabic and Hebrew are written right-to-left, unlike English which is left-to-right). Since interaction techniques are often organized together, such as in dialog boxes or forms (see Chapter 9), these will be affected by the size changes. In addition, as discussed in Section 20.7, programmers who implement UIs using interaction techniques must do extra work to support their UIs working in different natural languages.

1.9 Organization of the Book

The next two chapters (Part I) provide an overview of desktop and mobile devices, focusing on the systems that had significant impact on interaction techniques. Following that are 15 chapters (Part II) each focusing on a particular class of interaction techniques. Part III covers general concerns that are relevant to all interaction techniques, including models, implementation, and legal issues. Part IV contains interviews with 16 inventors of interaction techniques, discussing how they came to be. These inventors were the people who gave guest lectures to my class and are mostly my personal friends and acquaintances. Since these are based on those lectures, I was only able to include people who agreed to talk and also agreed to have their lecture recorded (which unfortunately eliminated May-Li Khoe's talk about the Apple Force Touch). Finally, Part V contains future work needed for interaction techniques and conclusions.

PART I

HISTORY AND BACKGROUND

All the individual interaction techniques described later in this book were developed with certain hardware and contexts in mind. Therefore, it is important to know about the kinds of computers and other devices that these interaction techniques were designed for. This part provides a brief history of the kinds of computers that have been in use and the important systems for which the interaction techniques were developed. Whereas some of these may be familiar, others are more obscure except to historians of technology and user interfaces, but all are important in their influence on the evolution of interaction techniques. I have also included a discussion of different styles of user interfaces since these also have had a big impact on the interaction techniques used.

This part is divided into two chapters—Chapter 2 on regular computers, such as desktops and laptops, and Chapter 3 on mobile devices. Readers who are interested in the history of user interfaces should find these chapters interesting, even apart from interaction techniques. There are also explanations of many terms and concepts that will be used throughout this book (and which are useful for any HCI person to know, in general).

CHAPTER 2

History of Desktop Devices

2.1 Introduction

This chapter provides a brief history of "regular" computers to form a basis for understanding their interaction techniques. Since we are going back in history, this includes the original large room-size computers, from *mainframes* and *minicomputers* (defined in Section 2.2), to *personal computers* that fit on or under a desk, and then to the laptops and desktops that we use today. Separately, Chapter 3 discusses the history of *handheld* and *mobile* devices such as *tablets*, *personal digital assistants* (PDAs), and *smartphones*.

There are many books and articles that provide a more comprehensive history of computers, their user interfaces, and the people involved (e.g., Goldberg [1988], Press [1993], Myers [1998a], and Isaacson [2015]), so this chapter focuses on technologies and systems that I think are important for understanding the origins and development of interaction techniques.

2.2 Computer Hardware

This section summarizes the key aspects of the hardware for computers as it relates to interaction techniques. First is the computer itself, and then the various kinds of displays used for the output from the computer to the users along with the input techniques and when they were developed.

2.2.1 Mainframes

The original computers were enormous machines (called *mainframes*) that took up whole rooms. Any users of these computers had to be programmers or operators [Grudin 2017], using whatever programming mechanism was available. In some cases, users of the time would flip switches (like on the *Colossus* computer from 1943–1945) or move wires and get feedback from lights and gauges on the machine (which therefore count as physical interaction techniques). For example,

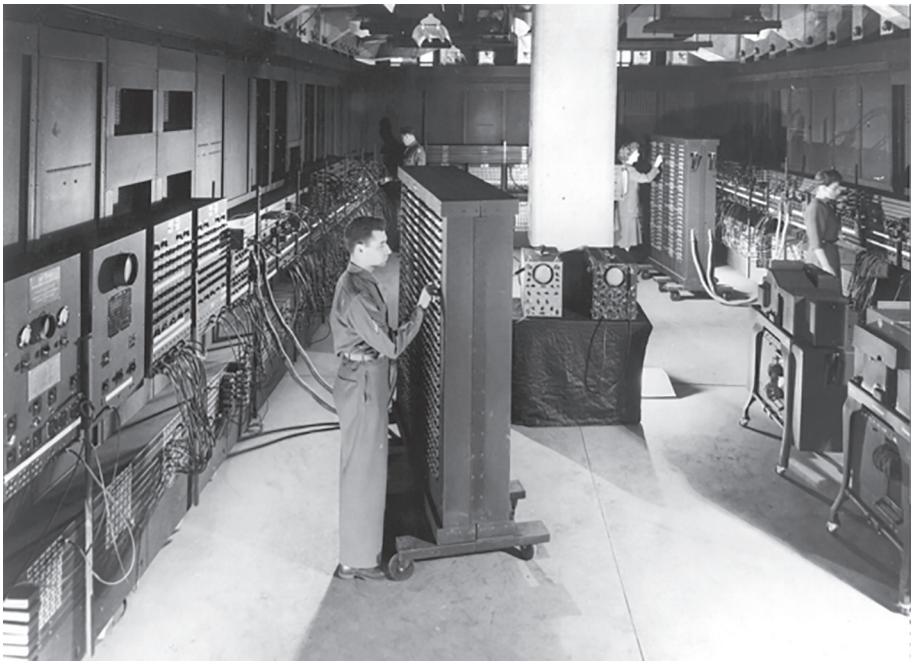


Figure 2.1 The ENIAC from 1945 [Rose 1946]. Source: Unidentified US Army photographer, public domain via Wikimedia Commons. https://commons.wikimedia.org/wiki/File:Classic_shot_of_the_ENIAC.jpg.

see Figure 2.1, which shows Cpl. Irwin Goldstein (foreground) setting the switches of the ENIAC computer in 1945 [Rose 1946]. It also shows some oscilloscopes with their screens that were used for hardware debugging. Various kinds of displays that could display graphics and text were also connected to mainframes (see Section 2.3).

Mainframes were often programmed by *punched cards* that the programmer had to punch using a separate card punching machine. Then, the computer would run the program and print out any results on a paper printer, which the user would get back hours or even days later. This form of "interaction" was still in use around 1976 when I created the punched card deck in Figure 2.2 for an undergraduate research project at MIT.

A next important step was *timesharing*, which means that the big computer was shared by multiple people who connected to it typically through a telephone wire. This was the main way people used computers from the 1960s until the 1980s. The hardware of the computers was sometimes still room size, but there emerged *minicomputers* from companies such as Digital Equipment Corporation (DEC) that were closer to refrigerator sized. Either way, the interactions were the same. Early users

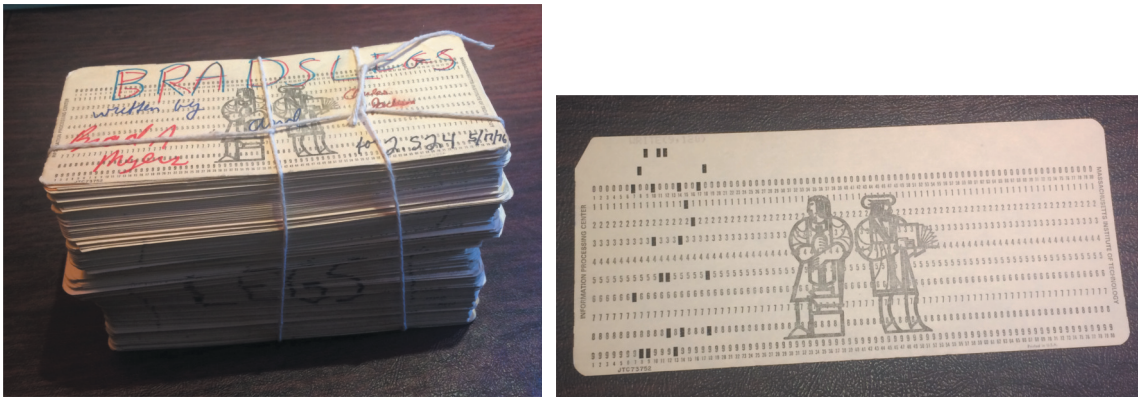


Figure 2.2 Punched cards I used at MIT in the 1970s. Sources: Original photos by author.

had printing terminals like the *teletype* machine I used in high school (Figure 2.3 shows a similar model). Users could enter programs by typing on the terminal or by reading in a paper tape (see Figure 2.4). More interesting from an interaction point of view was the development of terminals with screens that could be connected to timesharing systems (see Section 2.3) and supported much more interactivity. The interaction style for timesharing was mostly *command line* (Section 2.5.1.1) with some *form filling* (Section 2.5.1.2).

Figure 2.3 A teletype machine with a paper tape reader and punch from the 1960s. Source: Jamie, CC BY 2.0, via Wikimedia Commons. https://commons.wikimedia.org/wiki/File:Telex_machine_ASR-32.jpg.

Figure 2.4 Paper tape I used in high school in 1975. Source: Original photo by author.

2.2.2 Personal Workstations

Researchers at the Xerox Palo Alto Research Center (PARC) recognized in the early 1970s that the next generation of computers would be personal machines, where each person could have their own and would not need to share. They created the *Alto* computer (see Figure 1.9) as a stand-in to explore what the interactions on a personal machine might be like [Goldberg 1988]. This class of machines came to be called *personal workstations*, which generally were about the size of a dishwashing machine and had a mouse (Section 4.6) and relatively large bitmap display (see Section 2.3). Xerox tried to keep the Alto a secret for a while, and Alan Kay's early writings called it a *Personal Dynabook* [Goldberg and Kay 1976, Kay 1977], but eventually Altos were publicized and shared with research universities. Later, Xerox released the Xerox Star in 1981, described below in Section 2.4.4.

Meanwhile, a number of commercial companies built personal workstations for sale. An early kind (starting about 1979) were *Lisp Machines*, from companies like LMI (for Lisp Machines Inc.) and Symbolics, designed to run the programming language *Lisp*, invented by John McCarthy in 1958, which was popular in the 1960s–1980s especially for AI and other experimental programming [Sheil 1983]. Aiming for a broader market were three top workstation companies starting about 1982: Sun Microsystems, a spinoff from Stanford, Three Rivers Computer Corporation (PERQ computer), a spinoff from Carnegie Mellon University (CMU), and Apollo, a spinoff from MIT. I worked for Three Rivers Computer Corporation (later renamed PERQ Systems) from 1980 through 1984 and helped developed the software for their computer, which was called the *PERQ* (see Figure 2.5).

Later, bigger players like IBM, DEC, and Hewlett-Packard (HP) released their own personal workstations. These machines typically cost about $10,000 to $30,000 each and were intended for scientists, engineers, programmers, and other high-value white-collar workers for whom the increased productivity of an expensive personal computer could be justified. At the high end, Silicon Graphics Inc.

Figure 2.5 PERQ computer, monitor, and keyboard. It also had a pointing device. This was the first model of the PERQ, dating from approximately 1981. Source: Scan of original PERQ computer advertising brochure.

(SGI) was founded in 1982 and sold workstations that were designed for fast graphics processing. These were popular with animation and special effects companies such as those who made early computer-generated imagery (CGI) movies and commercials, such as Lucasfilm. A variety of software ran on these workstations, including proprietary operating systems and window managers [Myers 1984] and variants of the Unix operating system like SunOS with proprietary commercial window managers, and research window managers like Andrew [Palay et al. 1988] (see Section 2.4.12). Eventually, most workstations converged on using the "X" window manager after 1987 (see Section 11.3).

2.2.3 Personal Computers

Meanwhile, as the technology shrunk, it became possible to build personal machines that were much smaller and cheaper, which were generally called *personal computers* or *PCs*. Key competitors in this area have been the IBM PC running Microsoft Windows (see Section 2.4.8) and the Apple Macintosh (see Section 2.4.7) since the mid-1980s. Note that PC confusingly sometimes refers to the whole class of hardware and other times it means only the machines that run Microsoft Windows. These machines are typically divided into *desktops* and *laptops*, all of which have a screen, keyboard, and pointing device like a mouse or trackpad, and pretty much the same interaction techniques.

2.2.4 Other Devices

There are a wide variety of other hardware that support interaction techniques, including consumer electronic devices (like TVs, radios, automobile dashboards,

cable set-top boxes, etc.), factory equipment, mobile phones, tablet computers, computerized watches, and so on. There are not consistent names for all these classes of devices, so the various names and categories will be introduced as the devices come up. Many of these are further described in Chapter 3.

2.3 Display Technologies

The progression of display technologies is important for their impact on the kinds of interaction techniques that are possible. As mentioned above, early computers had gauges, lights, and oscilloscope screens (e.g., Figure 2.1). The oscilloscope, which dates back to the 1930s, evolved into the early graphics screens, such as used in Sketchpad (Section 2.4.1). In an oscilloscope, a beam of electrons is bent by magnets (called *deflecting coils*) to draw the desired picture on the surface, which has a coating that fluoresces (glows) where the beam hits it (see Figure 2.6). Thus, if a square is to be drawn, the beam moves to one point on the square, turns on, draws the outline of the square, turns off, and then goes back to the beginning and repeats, as fast as it can. The spot that is glowing on the screen fades away after the beam leaves. This style of display is called a *vector scan* since the beam traces (scans) the lines (vectors) of the picture. What will be displayed is defined by a *display list* (the list of line segments or vectors to be displayed) and is stored in the display hardware. An interesting property of vector scan displays is that, as the image gets more complicated, it takes the beam longer to trace it out, and eventually the screen starts flickering because the fluorescent glow dies out before the beam starts over. The computer programming interface to a vector display was typically to fill and edit the display list.

Television screens work very differently. This kind of screen, which is called *raster scan*, also used a cathode-ray tube (CRT) as in Figure 2.6, but this time the beam starts at the top left of the screen, sweeps across a horizontal line, goes back to the left edge down one line (or down two—many early screens alternated lines), sweeps across the next line, and so on. Thus, with raster scan the beam sweeps continuously independent of the picture, and the picture results from turning the beam on and off at the right times as it sweeps across. For old-fashioned analog TV, the signal from the antenna that goes to the CRT continuously tells the beam how bright to be at each spot. In contrast, the computer interface to a raster scan display is typically a section of memory that holds the image to be displayed, and there is hardware in the display to scan through the memory to control turning on and off the beam.

The first kinds of raster scan displays for computers were display *terminals*, which date back to 1969. These typically divided the screen into small character-size rectangles, so they are also called *character displays*. For example, the popular

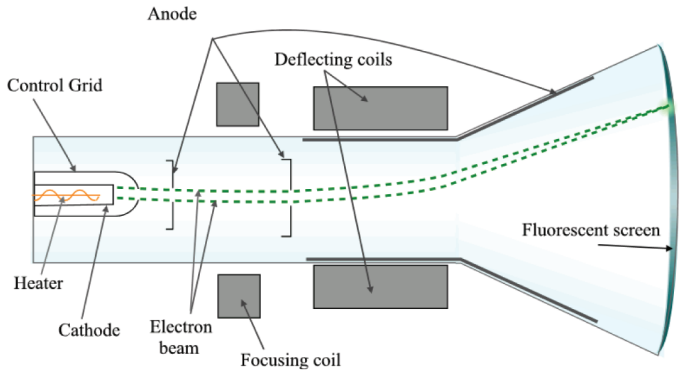


Figure 2.6 Cathode-ray tube. Source: Theresa Knott, CC BY-SA 3.0, via Wikimedia Commons. https://commons.wikimedia.org/wiki/File:Cathode_ray_Tube.PNG.

DEC VT100 terminal could display 24 lines of 80 characters each (see Figure 2.7). The terminals would typically connect to timesharing computers. The screen mimicked a printer, with new content appearing at the bottom and the rest of the content scrolling up, but the location where new text appeared, called the *cursor*, could be moved to any character location. Later, terminals supported a small set of graphic symbols like various lines and tiny pictures, which could be displayed in each character position. These displays were often used to support command line (Section 2.5.1.1) or form fill-in (Section 2.5.1.2) UIs. It was popular for the text to

Figure 2.7 DEC VT100 display terminal. Source: Jason Scott, CC BY-SA 2.0. https://commons.wikimedia.org/wiki/File:DEC_VT100_terminal.jpg.

be displayed in green on a black background (e.g., on the Apple II monitor from 1983), which explains why the text is green in the *Matrix* movies.

As PCs took hold, such as the Apple II, the Commodore Pet, Microsoft DOS, and so forth, they started with a screen that looked like a terminal, usually still 24×80 characters. When windowed systems appeared, often this kind of interface was still provided as a *console window* or *terminal window* and these all supported the same kinds of UIs as terminals.

The more interesting kind of raster scan displays supported access to the individual *pixels* (dots) of the screen. These were typically called *bitmap displays* since the computer controlled the display by using a section of the memory to map what each bit of the screen should show. The Alto (Figure 1.9) was one of the first to demonstrate the power of this technique, which was then used by most other workstations (see Figure 2.5). The first generation of these displays used *monochrome* displays, which means that each pixel could either be on or off (typically black or white), with no support for grays or colors. Thus, each pixel could be represented in the computer by a single bit (with the value of 1 or 0). To show different levels of gray, *halftoning* was used, which is a technique where alternate pixels are turned on and off, for example, every-other-pixel for half-gray, every fourth pixel for ¼ gray, and so on. Eventually color screens took over, where there were 8, then 16, then more bits for each pixel. These are sometimes called *pixmaps* since they have more than one bit per pixel. The next change was to flat screen technologies like liquid crystal displays instead of CRTs, which enabled both smaller and larger devices since they are much thinner, lighter, and need less power.

2.4 Important Systems

Most of this book discusses the evolution of individual interaction techniques as they changed through time. However, it is useful to start with an overall discussion of the *systems* in which those interaction techniques appeared since many of these systems introduced *multiple* interaction techniques. Therefore, this section briefly describes in chronological order many of the systems that will be referenced in the later chapters.

2.4.1 Sketchpad, 1963

Sketchpad [Sutherland 1963] was an enormously influential early system that was developed by Ivan E. Sutherland as his Ph.D. thesis from MIT in 1963. It ran on the experimental TX-2 computer at MIT's Lincoln Labs and was developed with support from the US Air Force and the US National Science Foundation (NSF) (see Figure 2.8 and the Sketchpad video [Sutherland 1964]). The user interface

Figure 2.8 Ivan Sutherland's Sketchpad system from 1963 being demonstrated by Timothy Johnson, who created the 3D extension to Sketchpad [Johnson 1963]. Source: Still from the Sketchpad video [Sutherland 1964].

for the TX-2 provided many physical switches and knobs as well as the lightpen (described in Section 4.3) to point at the vector screen (see Section 2.3). Sutherland was the first to demonstrate many concepts that we take for granted today, including directly pointing to objects on the screen to select and operate on them (now called *direct manipulation*, see Section 2.5.1.3) and providing interactive incremental feedback as the user is creating new objects in the form of a constantly changing shape (now called *rubber band lines*, see Section 10.6). Sketchpad let the users define *constraints* on the objects that it then enforced, such as that a line should stay attached to a circle or that lines should stay vertical or horizontal. The individual objects or the whole drawing could be arbitrarily scaled bigger and smaller using knobs (the first *zoomable user interface*, see Section 11.11.1.1).

Sketchpad also demonstrated a form of object inheritance now called the *prototype-instance model*, where the user can define a shape as a *prototype* and then use multiple *instances* of that shape. If the user later goes back and edits the original prototype, then the computer automatically changes all the instances to match (which is different from a *copy*, where any of the objects could be edited with no impact on any other copy). This form of inheritance is now widely used in languages such as JavaScript.

Later additions to Sketchpad included 3D drawing and manipulations [Johnson 1963], including hidden-line elimination, and the first interactive *visual*

programming system [Sutherland 1966] (which means that the program itself is defined using graphics [Myers 1986a]), created by William Robert "Bert" Sutherland, who is Ivan Sutherland's brother.

2.4.2 NLS, 1968

The next important system in the history of interaction techniques was called *NLS*, for the "oN-Line System" from the Stanford Research Institute (SRI). Led by Doug Engelbart, this group invented the computer mouse in 1967 [English et al. 1967]. The original version was made out of wood (see Figure 2.9(a)) and was designed to be used much as mice are today—rolled on the desk with the dominant hand. A *cursor* or *pointer* (see Chapter 4) moves on the screen to show where the user is pointing. The left hand was intended to be used on a five-button keypad (see Figure 2.9(b)), or else both hands could be on the keyboard.

SRI used the mouse in the NLS system, which is considered to be the first demonstration of a system where multiple people could collaborate in real time on a shared document. This was shown on December 9, 1968, at the Fall Joint Computer Conference in San Francisco in what is often called "The Mother of All Demos," which was filmed and is now available online [Engelbart and English 1968]. There is also a written description of the system [Engelbart 1988].

(a) (b)

Figure 2.9 (a) The original wooden mouse. (b) The keypad, keyboard, and mouse in use in NLS. Sources: (a) Courtesy SRI International, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=17294412. Image courtesy of SRI International, (b) Courtesy SRI International, from https://www.computerhistory.org/revolution/input-output/14/350/1882.

2.4.3 Xerox Alto, 1970s

Inspired by the demonstrations of the Sketchpad and NLS systems, the Xerox Corporation, foresightedly realizing that computers would eventually be important for knowledge work by office workers, created a research lab in 1970 called the Xerox Palo Alto Research Center (Xerox PARC). They hired several people from SRI who had worked on NLS, who brought over their mouse and keypad (Figure 2.9), along with an incredible collection of talented and creative people who invented many of the interaction techniques we use today. As mentioned above, they created their own computer, the Xerox Alto (Section 2.2.1 and Figure 1.9), and they improved the hardware for the mouse (see Section 4.6). In addition (unrelated to interaction techniques), Xerox PARC invented laser printing and the Ethernet local area network. In software, Xerox PARC also invented many of the concepts now used to *implement* user interfaces and their interaction techniques, such as object-oriented programming and the model–view–controller architecture, described in Chapter 20.

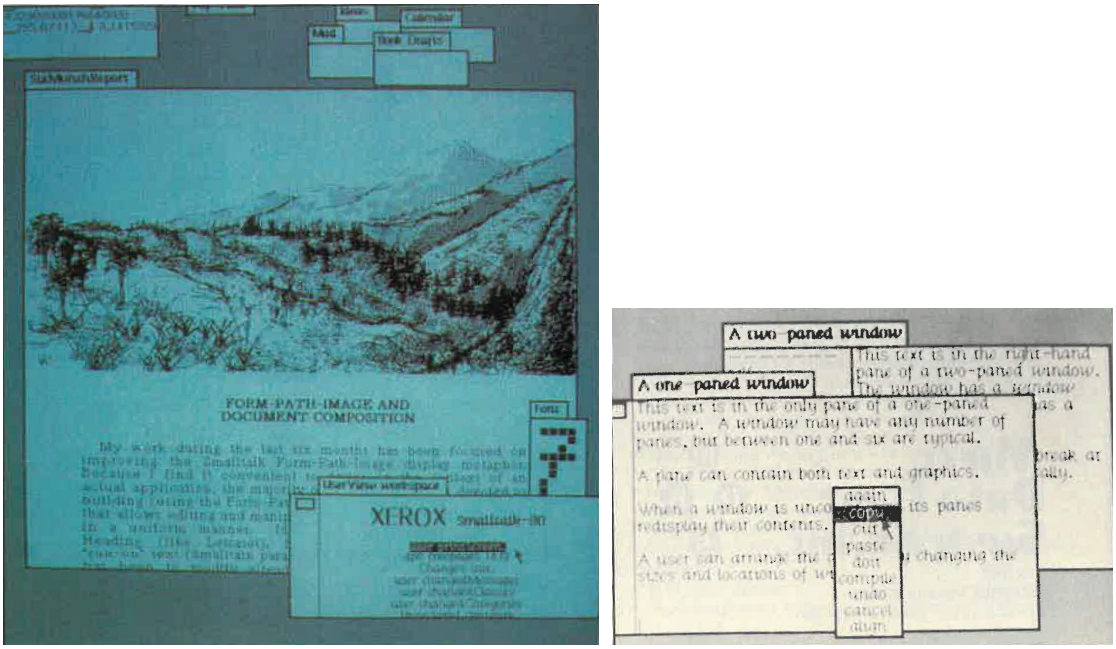
On the Alto, there was no operating system as we think of it today. Instead, each application took over the entire computer and defined its own entire user interface. Since there were no standard ways of interacting and concepts like trying to give a user a consistent look and feel hadn't been invented yet, this enabled the researchers to explore a wide range of interaction techniques.

Much later, in the 1990s, Xerox PARC invented many concepts of what is now called *ubiquitous computing*, described in Section 3.4 and Chapter 13.

2.4.3.1 Smalltalk, 1976

One of the early hires at Xerox PARC was Alan Kay, who had proposed the idea of overlapping windows in his 1969 doctoral thesis [Kay 1969]. This idea was first fully implemented in Kay's *Smalltalk* programming system built at Xerox PARC. Smalltalk was the first fully object-oriented programming language, where all the code was divided into classes in an inheritance hierarchy, and was enormously influential on programming languages and how interaction techniques are implemented (see Chapter 20).

The first published version of the language was in 1972 [Goldberg and Kay 1976], but the Smalltalk programming environment, which appeared in 1976, was also enormously influential in inventing and publicizing many interaction techniques. This is the version that I used when I was an undergraduate intern working in the Smalltalk group in the summer of 1977. I used all these interaction techniques to edit Smalltalk code to create a network stack (TCP/IP). My Smalltalk code



(a) (b)

Figure 2.10 (a) A Smalltalk screen. (b) A close up of two windows, with multiple panes, a scrollbar (on the left), and a popup menu. Source: From Tesler [1981], pp. 96, 102.

ran so slowly that you could see the individual packets moving through the system (the real networking code used by Smalltalk at the time was written in the underlying machine code, so it ran at an acceptable speed).

The Smalltalk-80 version was probably the most famous since it was described in a series of articles that appeared in *Byte* magazine in 1981 [Ingalls 1981, Tesler 1981] and a book [Goldberg and Robson 1983]. As shown in Figure 2.10, it supported multiple covered and paned windows, popup menus, scrollbars, use of copy-and-paste for editing, radio-buttons, and so forth, all of which will be covered in detail in later chapters of this book. Larry Tesler was a member of the Smalltalk group and had a big part in designing the environment, as described in his profile in Chapter 23.

2.4.3.2 Bravo Text Editor, 1974

The Bravo text editor was another early application on the Alto. It was developed by Butler Lampson, Charles Simonyi, and their colleagues around 1974, based on concepts they had started working on around 1970 while at Berkeley. Bravo is the

first text editor that provided *What you see is what you get* (*WYSIWYG*—pronounced "wiz-ee-wig") for formatting. This means that the display on the screen, including using multiple fonts, variable spacing, and flush-right borders (justification), matched the way the document would look when printed (see Figure 2.11). The designers of Bravo invented new ways to do text editing (see Chapter 8), along with new ways to scroll (Chapter 5), and ways to show multiple documents using paned windows (Chapter 11 and Figure 2.11). Charles Simonyi was hired by Microsoft to be the chief designer/developer for the original versions of Microsoft Word, which was first released around 1983.

2.4.3.3 Various Drawing and Painting Programs, 1970s

Another key interaction explored on the Alto was ways to draw and paint pictures on a computer (see Section 2.5.1.5 for a discussion of the difference between *drawing* and *painting* programs). For example, William Newman's Markup (1975) was

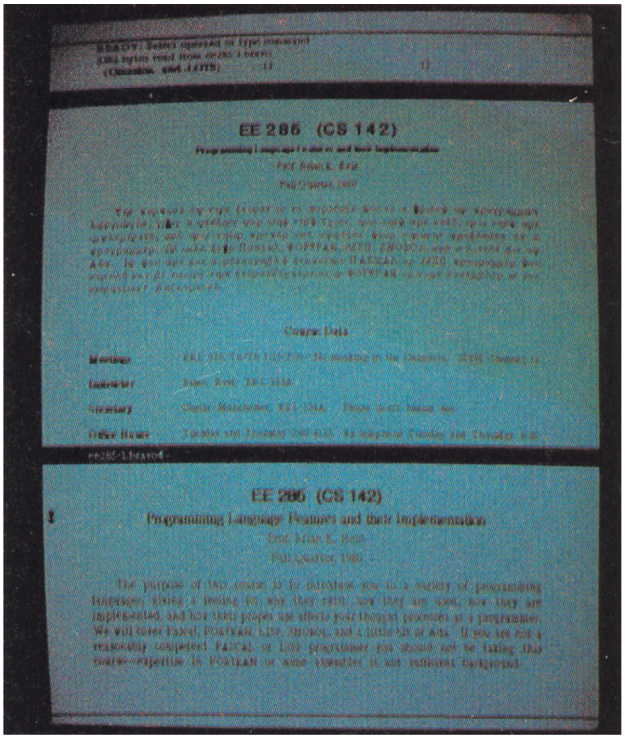


Figure 2.11 A screen showing two panes from the Bravo text editor. Source: Courtesy DigiBarn Computer Museum.

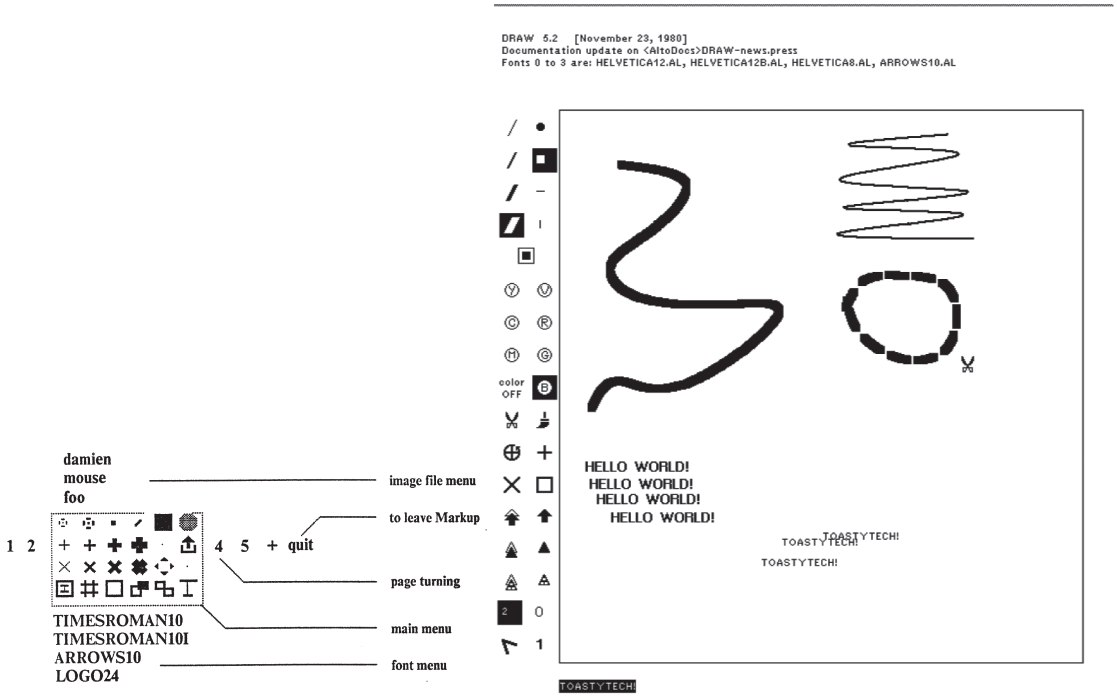


Figure 2.12 The Xerox Alto (a) Markup and (b) Draw screens. Source: (a) Markup manual. (b) Courtesy Nathan Lineback (ToastyTech), http://toastytech.com/guis/saltodraw.png.

the first drawing program for Xerox PARC's Alto. It supported drawing and editing lines and curves and featured a popup contextual menu of drawing operations that appeared at the cursor (see Figure 2.12(a)). Patrick Baudelaire's *Draw* program for the Alto (Figure 2.12(b)), which came out about the same time, introduced support for drawing curves and *inking*, where the user can draw any shape while holding the mouse down. PARC is also credited with having the first computer *painting* program, called *SuperPaint* by Dick Shoup (1974–1975).[1]

Around 1981, Xerox created a more powerful computer as a successor to the Alto, called the Dorado [Pier 1983], which was capable of rendering more elaborate graphics with a better user interface. Meanwhile, around 1979–1981, John

1. The menu for SuperPaint is here: https://web.archive.org/web/20150924090732im_/http://www.rgshoup.com/prof/SuperPaint/tns/SP_menu75tn.gif. Dick Shoup's personal page is archived here: https://web.archive.org/web/20150924090732/http://www.rgshoup.com/prof/SuperPaint/index.html.

Warnock, Chuck Geschke, and others at PARC developed a language for representing text and graphics sent to a printer, called Interpress [Warnock 2018].[2] The Gargoyle editor at PARC [Pier et al. 1988] from around 1985[3] was designed to take advantage of the capabilities of Interpress, and the faster processor of the Dorado, and introduced a number of innovations in graphics editing, including new ways to draw curves and snapping objects to other objects, for more accurate drawing [Bier and Stone 1986]. All of these applications had to invent their own interaction techniques for creating, selecting, editing, and deleting graphical objects (see Chapter 10).

2.4.3.4 Xerox Programming Environments, 1980–1981

Xerox PARC had different research groups using and designing different programming languages and their environments. One was the Smalltalk group described above (Section 2.4.3.1). Another was a group creating an imperative block-structured language called *Mesa* and its environments called the *Xerox Development Environment* (XDE) or Tajo (from 1981), and then the later Cedar [Swinehart et al. 1986] environment (see Chapter 11). (I used Mesa to implement my master's thesis system called Incense in 1979–1980 [Myers 1983].) A third environment was based on the *Lisp* programming language (see Section 2.2.2). Xerox called its version *Interlisp*, which had its own elaborate windowing environment called Interlisp-D [Xerox 1983] or DLisp [Teitelman 1979]. (I used Interlisp-D to implement my Ph.D. system called Peridot in 1985–1987 [Myers 1990c].) Interlisp-D's interaction techniques were mostly based on Smalltalk's, but some of the behaviors were different. (Eventually, the US government, which had funded the development of many different versions of Lisp, decided to force all the research groups to collaborate to create a single version of Lisp, called Common Lisp [Steele Jr. 1984], which I then used to create my first large-scale user interface toolkit called Garnet in 1987–1994 [Myers et al. 1990].)

2.4.4 VisiCalc, 1977

The first ever spreadsheet computer program was *VisiCalc* (1977-8), which was developed by Dan Bricklin (see Chapter 27) and Bob Frankston for the Apple II.

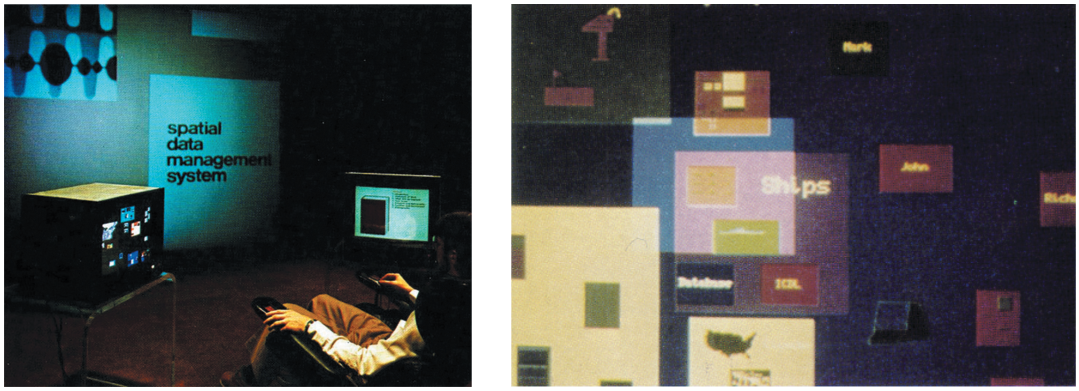
2. Warnock and Geschke became frustrated with Xerox and decided to form their own company, called Adobe Systems, in 1982, with the first product being PostScript, a successor to Interpress. They also created the Adobe Illustrator drawing program starting in 1985, first released in 1987, to take advantage of PostScript's capabilities and the Apple Macintosh user interface.

3. The "All the Widgets" [Myers 1990b] video shows a date of 1982 for Gargoyle, but Eric Bier, one of the creators, says that is incorrect, and it should be 1985.

The solver was based on a dependency-directed backtracking algorithm by Sussman and Stallman at the MIT AI Lab.

2.4.5 Spatial Data Management System, 1978

The SDMS [Bolt 1979, Herot et al. 1980] from the Architecture Machine Group ("ArchMac") at MIT (now the MIT Media Lab) was the first system to generalize the *zoomable user interface* shown in Sketchpad to support objects other than graphical shapes, including slides, electronic books, and movies, along with applications including a calculator, a contact list, and prototype phone dialer (see Section 11.11.1.2). As shown in Figure 2.13(a), in one instantiation, the user had an overview on a small nearby screen (called the *world view*), the full-size view was shown on a large projection screen in front, and another screen was available for a detailed view. The user navigated using physical joysticks embedded in the arms of the chair, and the monitors at the sides were touchscreens. I remember that Nicholas Negroponte, head of the Architecture Machine Group at the time, said that SDMS was the first system that showed an on-screen graphical calculator, and that people said that was silly and would never catch on! One of the members of the SDMS team, Chris Herot, left MIT and joined a company nearby called Computer Corporation of America (CCA), which created a semi-commercial version of SDMS primarily for military use (see Figure 2.13(b)), which was installed on the aircraft carrier USS *Carl Vinson* [Herot et al. 1980].



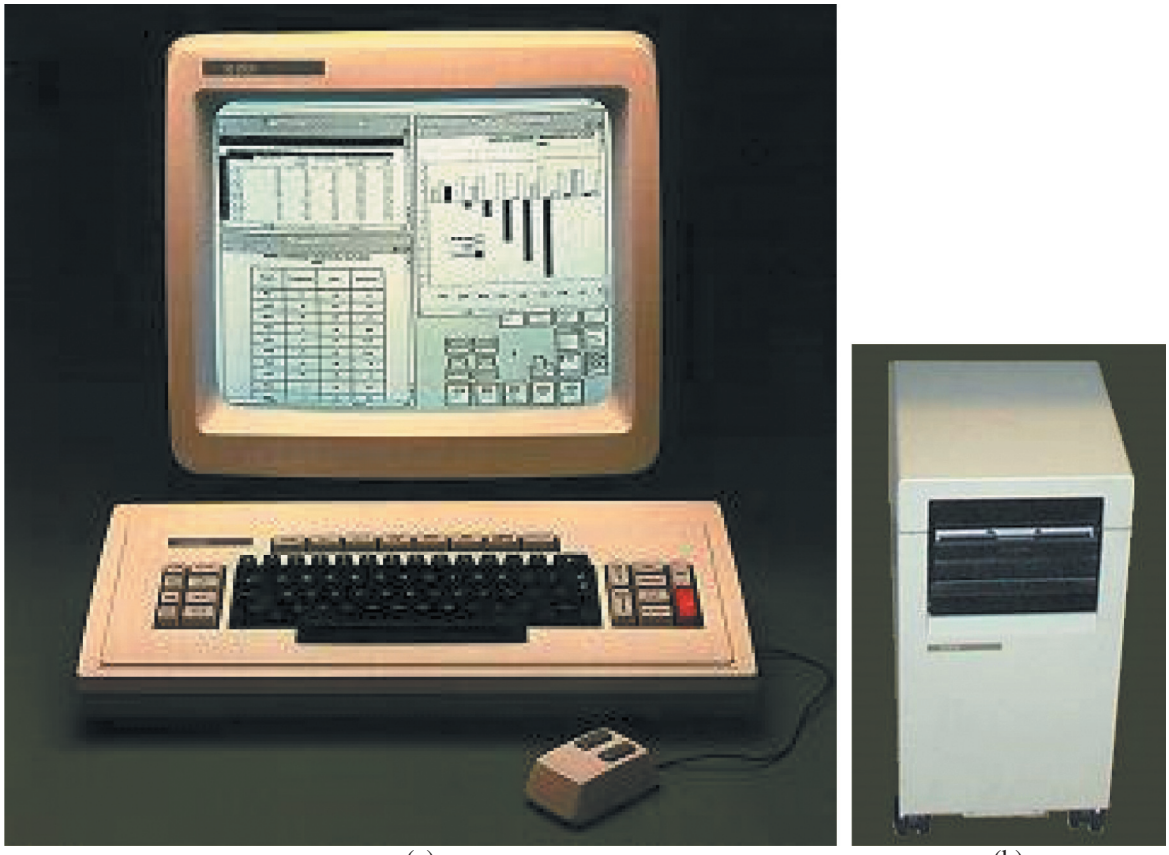
(a) (b)

Figure 2.13 The Spatial Data Management System (SDMS). (a) The overall configuration. (b) A close up of an overview ("world view") screen. Sources: (a) Bolt [1979]. Image courtesy of Stewart Brand. (b) Figure 6 from Herot et al. [1980].

2.4.6 Xerox Star, 1981

The Xerox Corporation wanted to leverage the inventions from PARC into a commercial product, so they formed another unit across the street from PARC that built the *Xerox Star*. This was an enormously influential and important machine, but it was not a commercial success. The Star system included a workstation with a large monitor and a two-button mouse (Figure 2.14) and software designed for office workers [Smith et al. 1982a, 1982b].

Key aspects of the Star interaction techniques included providing icons (little pictures) as part of a simulated desktop on the screen (the *desktop metaphor*). The idea for using icons in this way is attributed to David Canfield Smith (Chapter 24),



(a) (b)

Figure 2.14 The Xerox Star (a) monitor, keyboard, and mouse and (b) workstation. See also details of the keyboard in Figures 9.1 and 15.1. Source: Courtesy DigiBarn Computer Museum.

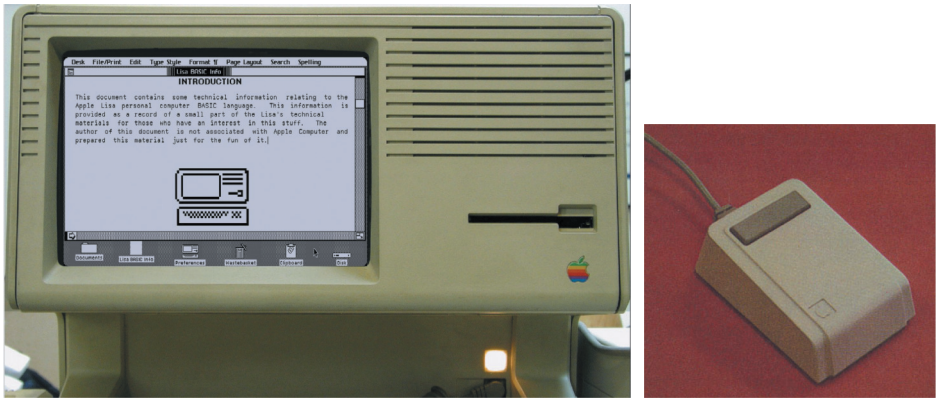
who coined the term in his 1975 Stanford Ph.D. thesis on Pygmalion [Smith 1977] (funded by ARPA and the USA's National Institute of Mental Health), and Smith popularized icons as one of the chief designers of the Star. The Star also included WYSIWYG text editing, chart and picture editing, and a full set of on-screen menus, radio buttons, property sheets, scrolling, and so on, all of which will be described below in the relevant chapters, as well as in Chapters 24 and 26.

A key barrier to the uptake of the Star was its price. Each machine cost about $10,000 (about $32,000 in 2023 dollars), but the Star was meant to operate as part of a whole system, so companies needed to also buy Xerox file servers, printers, and networking technologies. There was also the issue that at the time executives were not used to typing for themselves since they had secretaries to do that (the conventional wisdom of the time was that executives would never type). So executives did not want these machines for themselves, and they were too expensive to give to their secretaries. A further problem was that even though using a Star was much faster than using any kind of competing techniques of the time, users still experienced delays and saw the busy symbol on the screen a lot (see Chapter 6), which caused operations to *feel* slow.

2.4.7 Apple Lisa (1983) and Macintosh (1984)

The next important developments in interaction techniques come from Apple. The story, which is well-covered by many other books and even movies, is that Xerox was worried about its Star product not succeeding and wanted to invest in Apple as a backup plan before Apple became a public company. Steve Jobs made a deal to allow Xerox to invest in exchange for Apple getting to see the Xerox PARC inventions and having a free license to use anything from Xerox. As a result, Steve and his team (notably, Bill Atkinson, Chapter 28) were given a demo of various Alto programs in 1979, including the mouse, Smalltalk with its overlapping windows, the Bravo editor, and so on. In 1980, Apple hired Larry Tesler (Chapter 23) from PARC to begin designing their own systems based on those ideas. A key difference from the Xerox approach was that Apple would create machines that were significantly less expensive and would work for individuals. The focus was also not necessarily on business users, as evidenced by Apple's 1984 slogan "The computer for the rest of us."

The first product to come out of this was the Apple Lisa in 1983. This introduced many of the interaction techniques we are now familiar with, including the trashcan icon on the desktop, pull-down menus at the top of the screen, Apple's style of dialog boxes, the one-button mouse, and many other UI innovations (Figure 2.15). It took amazing programming expertise to get all of this to work on a tiny, inexpensive machine.



(a) (b)

Figure 2.15 (a) The Apple Lisa, and (b) its mouse. Source: (a) Courtesy Nathan Linback (ToastyTech). http://toastytech.com/guis/lisa.html, (b) courtesy Digibarn Computer Museum.

More famous was the release of the Apple Macintosh (Figure 2.16) the following year (1984), which used many of the Lisa's interaction techniques but in a smaller and cheaper form factor. The original Macintosh did not have a hard disk, just a single floppy, and it only had 128K bytes of memory. In order to make this work,

Figure 2.16 Original 1984 Macintosh.

much of the code that was used in the Lisa had to be rewritten for the Macintosh in the low-level assembly language of the hardware to make it small and be fast enough (see Section 28.2). The Macintosh was introduced to the world through a famous Apple Super Bowl ad in 1984 created by Ridley Scott [1984].

The Macintosh continues to be a key Apple product, and it has evolved significantly over the years, with better hardware and user interface enhancements. When Steve Jobs came back to Apple from NeXT in 1997 (see Section 2.4.11), he brought new underlying technology and some interface changes, but the overall interaction techniques remained surprisingly stable. Recently, some aspects of the Macintosh user interface have been changed to more closely match the iPhone (iOS) designs (Section 3.18).

2.4.8 IBM PCs (1981) and Microsoft Windows (1985)

Meanwhile, IBM had created its *personal computer* (PC) in 1981 (see Section 2.2.3), which supported a variety of operating systems but eventually mostly used Microsoft DOS (for *Disk Operating System*, also called MS-DOS). The DOS user interface looked like a character terminal (Section 2.3). Eventually, in response to the Star, Lisa, and Macintosh, Microsoft released Windows 1.0 in 1985 (see Section 11.3) as a DOS extension (Figure 2.17), which did not catch on. Eventually, Microsoft released Windows 2.0 in 1987 and Windows 3.0 in 1990, but it was Windows 3.1 from 1992 that started to really be widely used, in part because Windows 3.1 was the first version that was a real operating system (rather than just a DOS add-on), with support for virtual memory, protected multiple processing, and so on. See Section 11.3 for more on the history of Windows.

2.4.9 Apple HyperCard (1987)

Another important system is Apple's HyperCard, which was originally a free program for the Macintosh created by Bill Atkinson and released in 1987. The goal was to make it much easier for the general public to write programs for the Macintosh, hence its slogan: "programming for the rest of us," which is now called *end-user programming* (EUP) [Myers et al. 1992]. HyperCard (see Figure 2.18) introduced many UI innovations including tear off menus, pages that overlay each other, animated transitions between pages (like one can get now from Microsoft PowerPoint, Apple Keynote, or Google Slides), and a *tabbed notebook* look and feel (see Section 11.5). Although HyperCard was widely used as a prototyping tool (see Section 20.9) and to learn programming, it never took off as a platform for real applications.

Underlying the HyperCard UI and interaction techniques was a custom programming language called *HyperTalk*, which had some unique features. The syntax

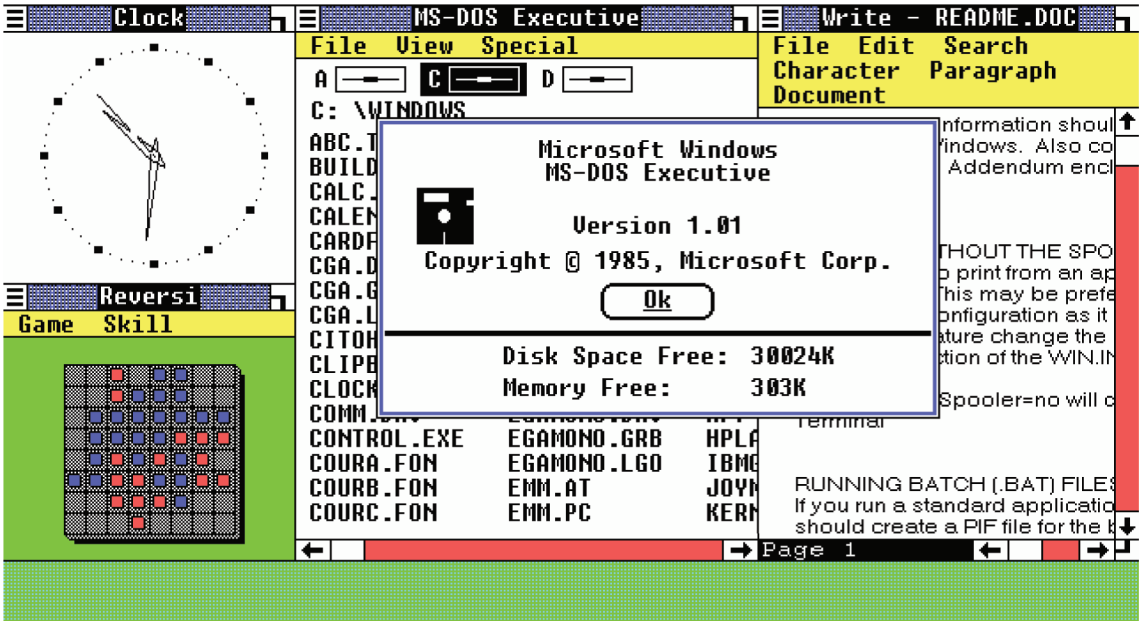


Figure 2.17 Microsoft Windows 1.0. Source: Screenshot by Remember the dot, used with permission from Microsoft, https://en.wikipedia.org/w/index.php?curid=8328551.

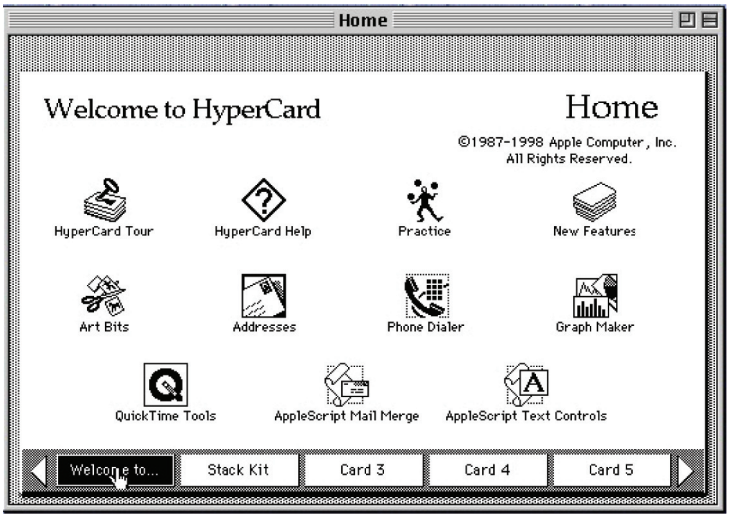


Figure 2.18 Apple HyperCard, 1987.

was more like English, with some flexibility to add words like "the" or "a," and it used a *prototype-instance model* (Sections 2.4.1 and 19.2).

2.4.10 Alternate Reality Kit, 1986

The *Alternate Reality Kit* (ARK) was a research project at Xerox PARC by Randall Smith [1986, 1987] to explore the limits of using a physical object metaphor for screen elements. The mouse cursor was rendered as a hand, and users "picked up" objects (including simulated balls but also buttons, sliders, and other UI elements, see Figures 2.19 and 10.18) from the screen by pressing and dragging them, which was rendered as moving them in 3D above the screen. Objects could be "thrown" with the mouse, which caused them to continue to move as if they had inertia. ARK was influential in its extensive use of the 3D look and feel, which was rendered using halftoning (Section 2.3) on a monochrome screen.

2.4.11 NeXT, 1985–1997

Shortly after the release of the Macintosh, Steve Jobs was forced to leave Apple, and he resigned on September 13, 1985. He then formed a new company called NeXT

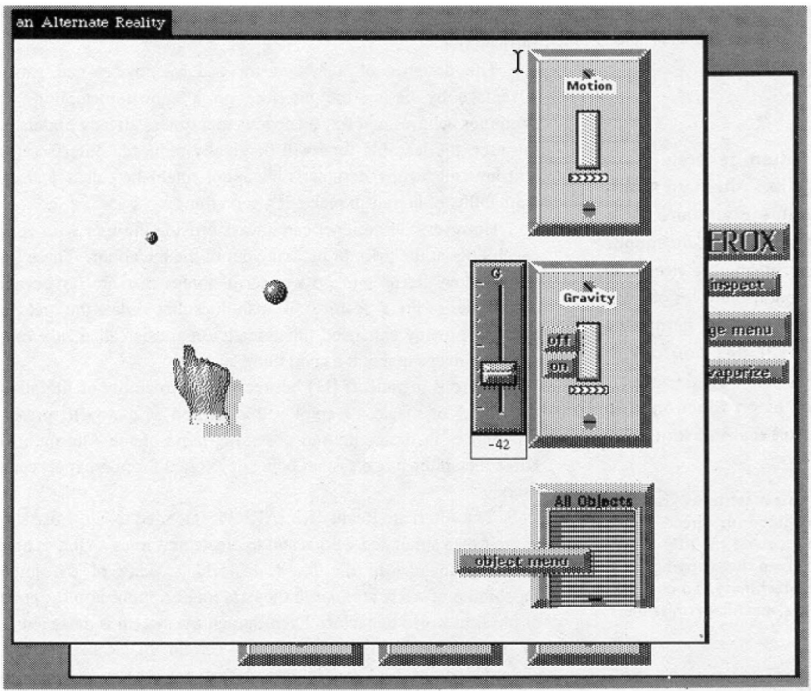


Figure 2.19 The Alternate Reality Kit showing the hand as the mouse cursor, and various objects that could be manipulated. Source: Figure 1 of Smith [1987].

that created workstation class machines (Section 2.2.2) for engineers and universities. I interviewed at NeXT in the spring of 1987, and they were so secretive about their product that they wouldn't tell me what I would be working on, and I had to do the interview in the front lobby. The product was announced in 1988, and Steve Jobs made the rounds of universities to show off the machine, including a visit to CMU on November 23, 1988, which I attended as a new faculty member.

The NeXT embodied a number of innovations in its hardware, UI, and interaction techniques. The computer itself was a black cube rather than a tall rectangle like other workstations. Color screens were still not affordable, but in order to make sure that the software didn't rely on monochrome graphics (like HyperCard), the NeXT used 2 bits per pixel, so it natively supported black, white, and two levels of gray and was easy to change to supporting color (see Figure 2.20). The look and feel of the interaction techniques were intentionally changed slightly from the

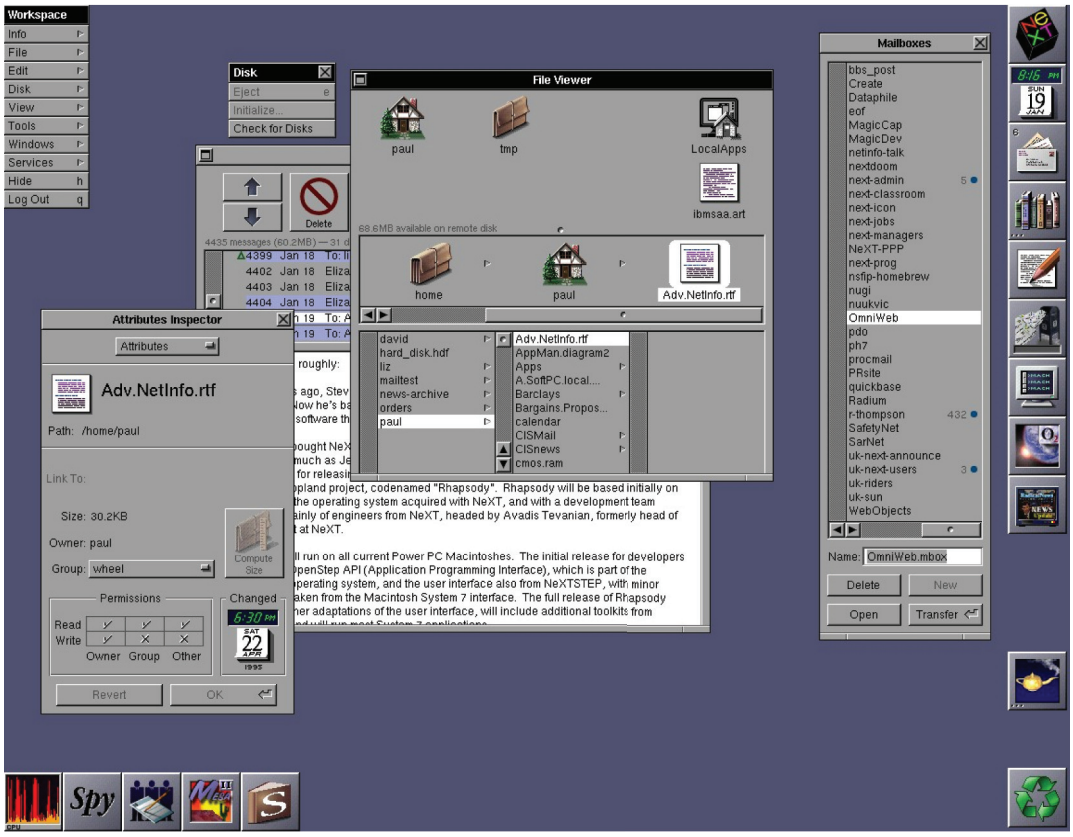


Figure 2.20 NeXTSTEP window system for the NeXT computer from around 1989. Source: By Source, Fair use, https://en.wikipedia.org/w/index.php?curid=34598118.

Macintosh (like the vertical scrollbars have both up/down arrow buttons together at the bottom) to try to avoid any copyright infringement lawsuits.

The implementation of the UIs in software was also interesting. NeXT worked with Adobe to adapt PostScript as the way graphics and text would be rendered onto the screen, which was called *Display PostScript* [Myers 1988]. Creating the UI was centered around an interactive tool called the *NeXT Interface Builder* (Section 2.5.2.2), which allowed designers to create many aspects of the UI using the mouse without programming.

Although the NeXT looked cool (as shown throughout this book), it turned out to be too expensive and incompatible and was not a successful product. (I had one on my desk for a few years but found little use for it, and instead primarily used a Sun workstation for my work. However, a NeXT machine was used by Tim Berners-Lee to create the first web server in 1990.) In 1997, Steve Jobs was rehired by Apple, and the NeXT software was merged into Apple's, influencing the interaction techniques and especially the implementations that followed.

2.4.12 SunWindows, NeWS, Andrew, X, OpenLook, and Motif

As introduced in Section 2.2.2, an important class of computers from the 1970s through the 2010s were *personal workstations*. Originally, each company created their own operating system (usually a form of Unix) as well as their own window manager and UI toolkits. Most looked quite similar, with mostly cosmetic differences, although there were a few behavior differences in some systems, as discussed throughout this book.

A leading workstation company was Sun Microsystems, which had a number of influential window and UI systems over the years. Early on, they provided *SunWindows*, which contained a fairly ordinary windowing system with a collection of conventional widgets like buttons, menus, text input fields, and scrollbars [Myers 1988]. The systems from their competitors like DecWindows from DEC were similar. Later, in the late 1980s, Sun developed a window manager called SunDew [Gosling 1986b], later renamed NeWS (Network extensible Window System), that was based on PostScript. Unlike NeXT's Display PostScript (Section 2.4.11), NeWS allowed programmers to download PostScript code into the window manager to enable faster execution of small interactive programs, but this also resulted in security problems since the NeWS execution environment was not protected. In terms of UIs, NeWS (and Display PostScript) enabled non-rectangular windows and many other graphics effects, but was slow.

Another notable window system of this time was the *Andrew* system [Palay et al. 1988] from CMU and IBM that innovated a unique form of menu (Figure 7.17(b)) and an influential component architecture (Section 2.5.2.3).

As explained in Section 11.3, as third parties started to develop software to run on workstations, such as editors/formatters (e.g., *Interleaf* and *FrameMaker*) and graphics programs (e.g., from Wavefront and Adobe), they objected to having to edit their UI code to adapt to each vendor's unique toolkit API, so they formed a consortium with MIT to develop the X Window System [Scheifler and Gettys 1986] and UI toolkit *intrinsics* called Xt [McCormack and Asente 1988]. A goal of X and Xt was to allow each of the manufacturers to have their own branded look and feel (see also Section 20.5.1), which succeeded, but this still prevented third parties from having vendor-independent code (the low-level graphics calls were standardized by X, but the widget code like to display a menu still differed). The various UIs created by the vendors' programmers were also not very attractive. Therefore, two new competing consortiums were formed to define the widget layers, called OpenLook (Figure 2.21) and Motif (Figure 2.22). OpenLook's UI was created by Xerox (with Tony Hoeber at Sun) based on the original Xerox Star look and feel and was supported by IBM, Sun, and others, whereas Motif was primarily designed by HP and supported by DEC and others. Having two incompatible systems did not meet the industry's needs, so, eventually, Motif won out around 1993 and is still used today on Linux systems.

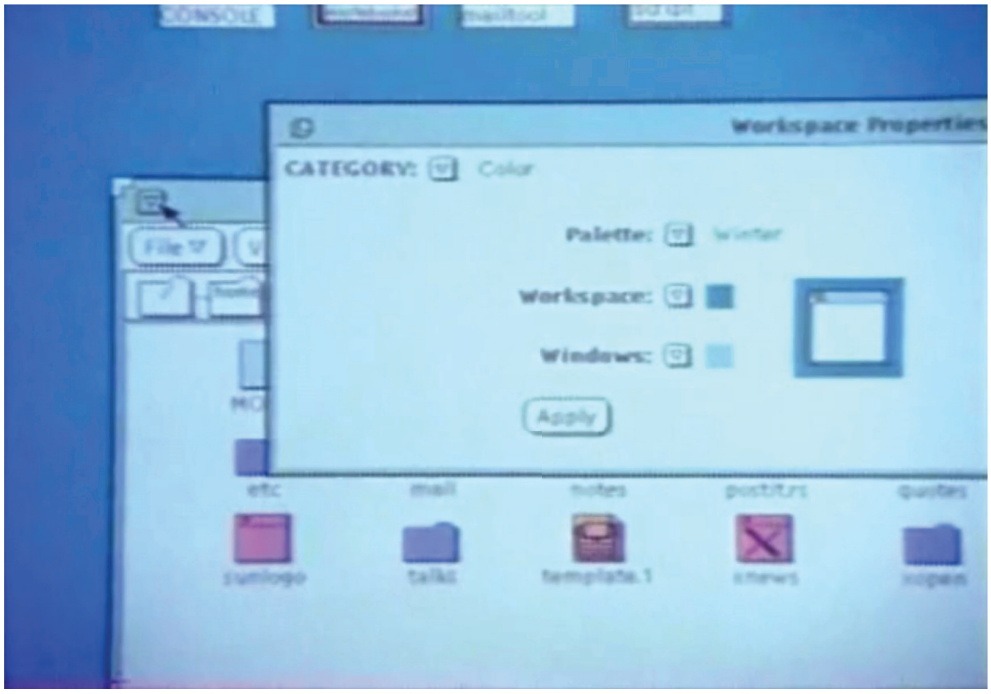


Figure 2.21 Portion of an OpenLook screen. Source: Myers [1990b] at 34:45.

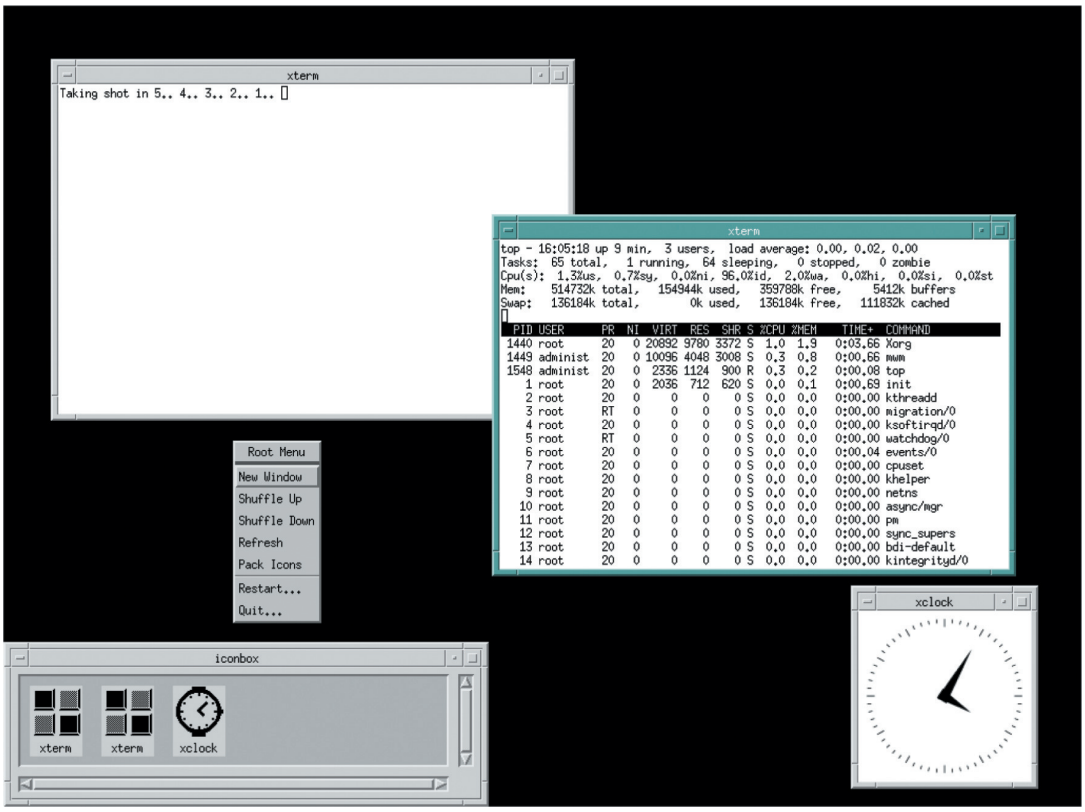


Figure 2.22 Portion of a Motif screen. Source: Huihermit—Own work, CC0, https://commons.wikimedia.org/w/index.php?curid=30556759.

2.4.13 World Wide Web

Section 2.5.1.8 below discusses hypertext in general, but it is useful to have a discussion here about the *World Wide Web* (*WWW* or Web) as an important system.

Tim Berners-Lee used the hypertext idea to create the Web in 1990 at the government-funded European Particle Physics Laboratory (CERN). Mosaic, the first popular hypertext browser for the WWW, was developed at the University of Illinois' National Center for Supercomputer Applications (NCSA). The WWW quickly spread as an easy way for researchers, universities, and eventually the general public and companies to provide and share information. From the beginning, it supported formatted text and pictures implemented in a language called *HTML* (for HyperText Markup Language) based on the Standard Generalized Markup Language from the 1980s, which in turn is based on markup languages that date back to the 1960s (see also Section 8.6.1 about text markup). Markup languages had long

had the concept of separating the style from the content, and this was applied to the WWW with Cascading Style Sheets (*CSS*), starting in 1996. However, CSS was implemented differently across different browsers and had slow uptake until CSS3, starting around 1999.

A variety of free and commercial *web browsers* run on the user's device (at first workstations, and later PCs and mobile devices) and communicate to remote servers that are identified by their Uniform Resource Locator (*URL*). Mosaic was the first popular hypertext browser for the WWW and was developed in 1993 at the University of Illinois' National Center for Supercomputer Applications. Next came Netscape Navigator (1994), Microsoft Internet Explorer (1995), Apple Safari (2003), Google Chrome (2008), and Microsoft Edge (2015), although there have been many others. A key reason that the WWW caught on at first was that it used its own communication protocol called the Hypertext Transfer Protocol (*HTTP*), which subsumed a number of widely used older and harder-to-use technologies like FTP for file transfers, Usenet newsgroups for discussions, and so forth. As discussed below in Section 9.7, the WWW was extended in 1993 to support data entry using forms, including adding to HTML the usual widgets—such as text and number entry fields, dates, URLs, checkboxes, radio buttons, and so on [Berners-Lee 1993, section 10]. The HTML5 standard was first released in 2008 and incorporated natively many features such as video, animations, and much more, which, along with JavaScript and CSS3, removed the need for other web languages such as Java or Flash.

To make actual applications on the Web requires more interactivity than was possible by just providing widgets in web pages, and a variety of approaches have been popular over the years. Sun Microsystems had developed the *Java* language and system to run in their digital television set-top boxes starting in 1991, but that application was not successful. However, Java worked well for the early Web and was released in 1996 as a way to provide *applets* in browsers. Java was largely replaced by the use of the *Flash* system. Macromedia Flash was first released in 1996 as a way to author animated and interactive content. In 2000, a programming language called ActionScript was released for Flash that made it possible to create a wide variety of content. To run Flash on a web page, the user had to install a plugin into their browser, and by 2005 Flash was the dominant media player plugin on browsers. That same year, Adobe purchased Macromedia, so Flash became an Adobe product and continued to dominate highly interactive content like games and video on the Web. However, the release of HTML5/CSS3/JavaScript with built-in capabilities that obviated much of what Flash offered in 2008, along with the refusal of Apple to allow Flash to run on the iPhone, precipitated Flash's rapid decline, and Adobe stopped supporting Flash altogether in 2017.

Flash and Java were completely replaced by the use of *JavaScript*, which was first released as a web language by the Netscape company in 1995. JavaScript can access and change the web page elements using the Document Object Model (DOM), which the browser creates from the HTML. Like other web technologies, JavaScript suffered from the *browser wars* of incompatibilities between different browsers, especially Netscape's Navigator and Microsoft's Internet Explorer, but by December 2009 there was a well-defined standard version and, importantly, very fast implementations of JavaScript in browsers, starting with Google Chrome, through the use of "just-in-time" compilation. This precipitated JavaScript's rapid rise to becoming the key language of the Web and one of the top programming languages overall.

Early in the WWW's history, it was recognized that it would be useful to access it on *mobile phones* (see Section 3.10), but the phones of the time only had small screens and slow data networks (Section 3.12), so the industry created a reduced version of HTTP and HTML called the Wireless Access Protocol (WAP), released in 1997. However, WAP had terrible usability [Nielsen 2000] and was not very successful. Internet access on mobile devices didn't really take off until smartphones with larger screens (e.g., the Apple iPhone in 2007, Section 3.18) and 4G transmission speeds (first released in 2008, Section 3.12) became widespread.

2.5 Brief Overview of Human–Computer Interaction Technology

An orthogonal way to summarize the history of interaction techniques, instead of going system by system, as above, or focusing on each interaction technique, as in subsequent chapters, is to look at the progress of each kind of interaction technology. This section[4] provides a brief summary of these HCI technologies, specifically showing how research at universities and corporate research labs had contributed to the creation of virtually all interaction technologies [Myers 1998a].

Figure 2.23 shows timelines for some of the technologies discussed in this section. Of course, a deeper analysis would reveal much interaction between the university, corporate research, and commercial activity streams. It is important

4. This section is adapted from my 1998 paper [Myers 1998a] that was developed with help from Stacey Ashlund, Meera M. Blattner, Keith Butler, Stuart K. Card, Bill Curtis, David E. Damouth, Dan Diaper, Dick Duda, Tim T.K. Dudley, Steven Feiner, Harry Forsdick, Bjorn Freeman-Benson, John Gould, Wayne Gray, Mark Green, Fred Hansen, Bill Hefley, D. Austin Henderson, Jim Hollan, Jean- Marie Hullot, Rob Jacob, Bonnie John, Sandy Kobayashi, T.K. Landauer, John Leggett, Roger Lighty, Marilyn Mantei, Jim Miller, William Newman, Jakob Nielsen, Don Norman, Dan Olsen, Ramesh Patil, Gary Perlman, Dick Pew, Ken Pier, Jim Rhyne, Ben Shneiderman, John Sibert, David C. Smith, Elliot Soloway, Richard Stallman, Ivan Sutherland, Dan Swinehart, John Thomas, Alex Waibel, Marceli Wein, Mark Weiser, Alan Wexelblat, and Terry Winograd.

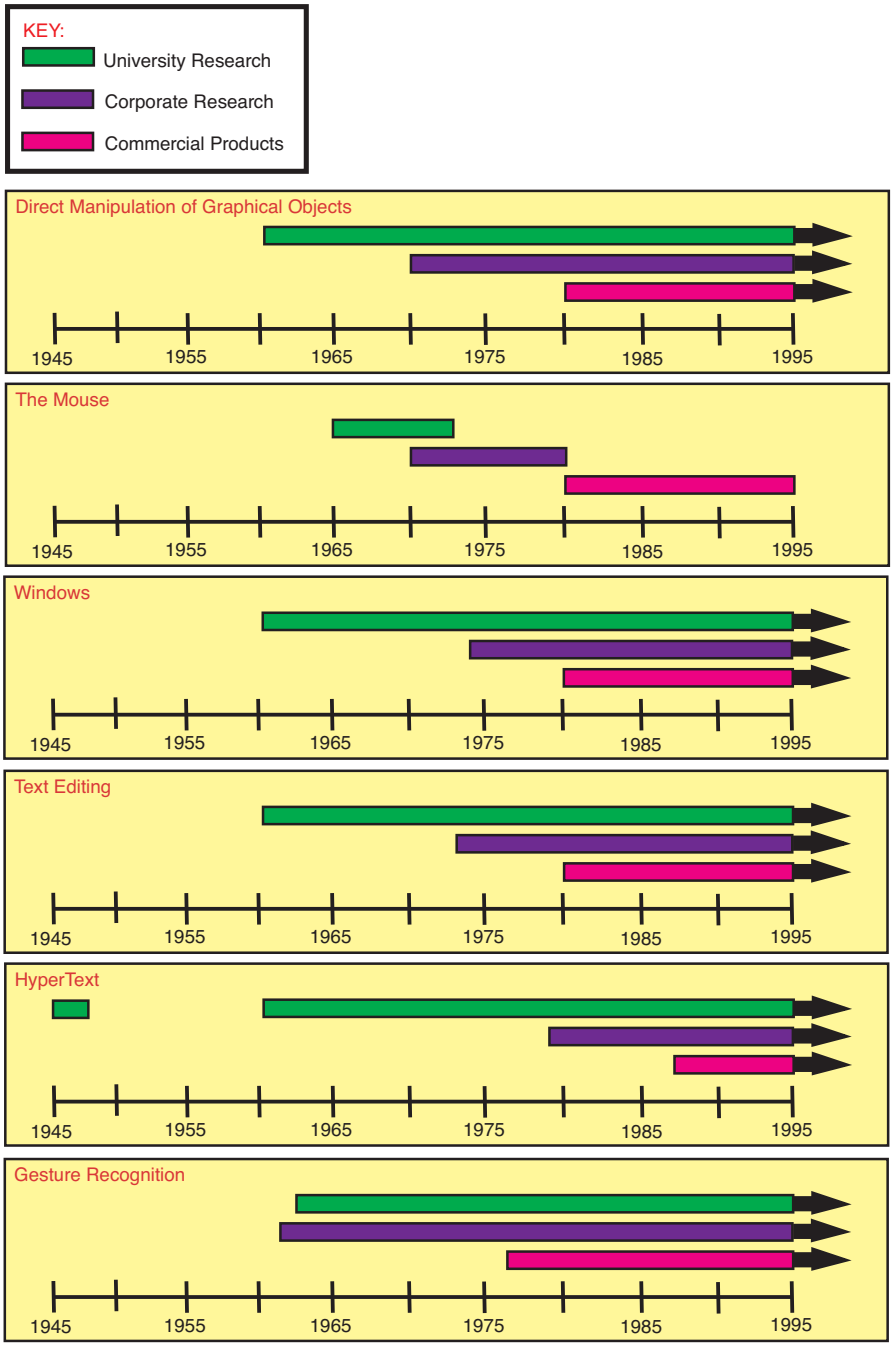


Figure 2.23 Approximate timelines showing where and when work was performed on some major technologies. Source: Original by author; previously appeared in Myers [1998a].

to appreciate that years of research are involved in creating and making these technologies ready for widespread use.

2.5.1 Interaction and Application Styles

2.5.1.1 Command Line Interfaces

Command line interfaces (CLIs) became the main way to interact with computers during the timesharing era (from the 1960s until the 1980s, see Section 2.2.1), first on printing terminals and then on character terminals (Section 2.3). It was also the usual way to interact with Microsoft DOS before Windows (Section 2.4.8). However, CLIs never went away and have been available on most computers in special windows, called the *console, terminal, cmd*, *shell*, or *DOS* window. For example, programmers still often prefer to use the CLI for applications like version control (like Git[5]), testing, or system management.

In a CLI, the user types a command followed by its arguments, and the system responds with the output of that command (see Figure 2.24). CLIs are almost always *prefix* (Section 10.2), meaning the command goes first, before the arguments. Long-running operations can often be canceled by hitting Control-C or other equivalents on the keyboard (see Section 15.4).

2.5.1.2 Form Fill-in

As soon as there were display terminals in the early 1970s (see Figure 2.7), computers began to be used to fill in *forms*, which are screens with places for users to enter data. Typically, the forms would be used to add, review, and edit database entries. The places where data go are called the *fields* of the form. Quite early on, there emerged standards for how users would interact with forms, such as using the *Tab* key on the keyboard to move from field to field, and many other recommendations for how to make this easier to use [Shneiderman 1987]. Form fill-in became a key interaction style on PCs, and products such as Dan Bricklin's DEMO (1985) showed how these could be made easier for programmers to design (see Chapter 26). On direct manipulation systems, forms are a type of *dialog box* (Chapter 9), and the text entry fields are their own widgets (Section 8.7). Now, many web pages are forms with fields that users fill in.

2.5.1.3 Direct Manipulation of Graphical Objects

The now ubiquitous *direct manipulation* interface, where visible objects on the screen are directly manipulated with a pointing device, was first demonstrated

5. Git was created and named by Linus Torvalds, who also created Linux. The name means an unpleasant person in British slang, and Torvalds is quoted as saying "I name all my projects after myself," according to Wikipedia.

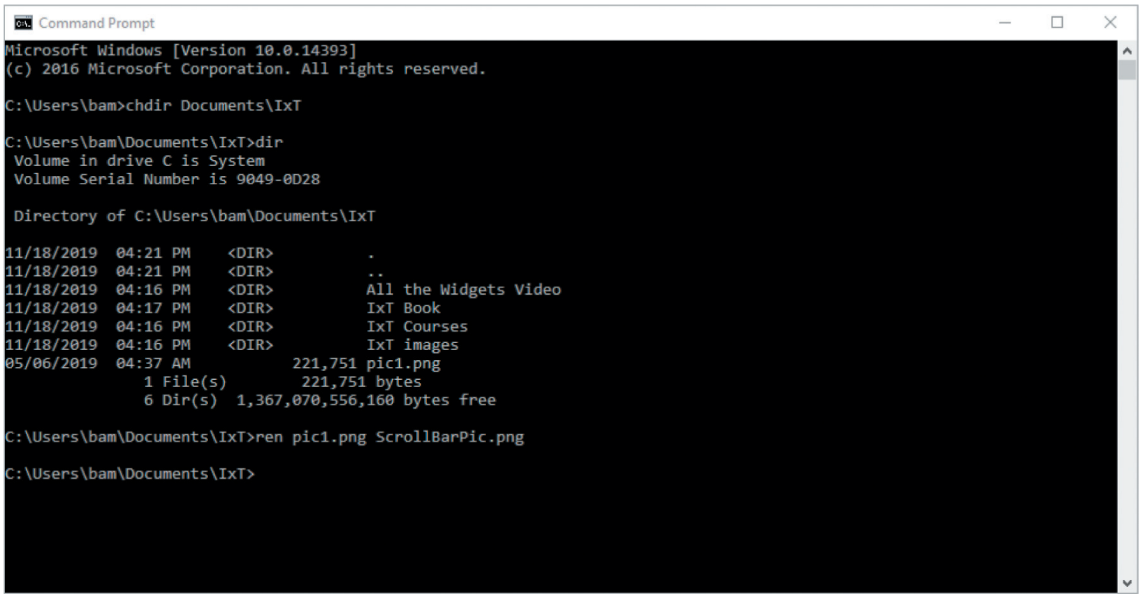


Figure 2.24 A command line interface is still available in Windows 10 in the "cmd" window. Here, the user has typed the commands "chdir" (change directory), "dir" (list contents of the current directory), and "ren" to rename a file. Source: Screen capture of Command Prompt on Windows 10, used with permission by Microsoft.

by Ivan Sutherland in Sketchpad in 1963 (Section 2.4.1) (see Figure 2.8). William Newman's Reaction Handler [Newman 1968b], created at Imperial College, London (1966–1967) provided direct manipulation of graphics and introduced "Light Handles" [Newman 1968a], a form of graphical potentiometer that was probably the first widget. Another early system was AMBIT/G (implemented at MIT's Lincoln Labs in 1968 and funded by the US Department of Defense's [DOD] Advanced Research Project Agency [ARPA], which also funded the development of the Internet and many other aspects of computing). AMBIT/G used, among other interface techniques, iconic representations, gesture recognition, dynamic menus with items selected using a pointing device, selection of icons by pointing, and moded and mode-free styles of interaction [Christensen 1968].

Many of the interaction techniques popular in direct manipulation interfaces, such as how objects and text are selected, opened, and manipulated, were researched at Xerox PARC in the 1970s (Section 2.4.3), including WYSIWYG editing in the Bravo text editor (Section 2.4.3.2) and the Draw drawing program (Section 2.4.3.3). The concept of direct manipulation interfaces for everyone was envisioned by Alan Kay of Xerox PARC in a 1977 article about the *Dynabook* [Kay 1977], which he later embodied in his Smalltalk system (Section 2.4.3.1).

The first commercial systems to make extensive use of Direct Manipulation were the Xerox Star (1981) (Section 2.4.3.4), Apple Lisa (1983), and Macintosh (1984) (Section 2.4.7). Ben Shneiderman at the University of Maryland coined the term *direct manipulation* in 1982, identified the components, and gave psychological foundations [Shneiderman 1983].

2.5.1.4 Window Managers

A *window manager* is a software package that helps the user monitor and control different contexts by separating them physically onto different parts of one or more display screens [Myers 1988], and a *window* is an area of the screen (typically a rectangle) through which the user interacts with one or more processes. A full discussion, including a history of window managers, is provided in Chapter 11, but briefly, having windows side-by-side (not overlapping) dates back at least to Engelbart's NLS in 1968 (Section 2.4.2). Overlapping windows are credited to Alan Kay, and they first appeared in 1974 in his Smalltalk system (Section 2.4.3.1). Commercially, the primary early examples were the Xerox Star in 1981 (Section 2.4.4) and then the Apple Lisa in 1983 and Macintosh in 1984 (Section 2.4.7). Microsoft Windows 1.0 was released in 1985, but it was Windows 3.1 from 1992 that began to catch on (Section 2.4.8). The X Window System, an international standard for use on all kinds of Unix (e.g., Linux), was developed at MIT in 1984 [Scheifler and Gettys 1986] (Section 2.4.12). The Xerox Star introduced the *desktop metaphor* in 1981 with icons. Mobile devices rarely support more than one window at a time, but often do use icons.

2.5.1.5 Drawing and Painting Programs

For creating graphics on a computer, it is important to understand the difference between *drawing* and *painting* systems (although many places will not be precise in their use of these terms). In a *drawing* system, graphical objects maintain their integrity after being drawn; in a *painting* system, objects are rendered into only pixels after being drawn such that the original object is no longer available. For example, in Microsoft PowerPoint and all other slideshow programs, along with Adobe Illustrator and many others, which are *drawing* systems, each graphical object (including text, lines, circles, etc.) can be selected any time after it is drawn and changed or moved. However, in MacPaint or Microsoft Paint, after an object is created, the user can only edit its pixels. Note that most painting programs are actually hybrids today (like Adobe Photoshop) and have some objects or layers, like text, that continue to be editable drawings unless they are explicitly *flattened* to be just pixels, with other items being rendered to pixels immediately.

Much of the current technology for *drawing* was demonstrated in Sutherland's 1963 Sketchpad system (Section 2.4.1). The use of a mouse for graphics was demonstrated in NLS (1965). Early drawing programs include *Markup* and *Draw* for Xerox PARC's Alto (Section 2.4.3.3). The first computer *painting* program was probably Dick Shoup's *SuperPaint* at PARC (1974–1975). The first widely used painting and drawing programs were MacPaint and MacDraw on the Macintosh (1984), which were designed to be easy to use. Adobe released the Illustrator drawing program in 1987 and the Photoshop image editing (painting) program in 1988, both first on the Macintosh.

2.5.1.6 Text (Document) Editing

In 1962 at SRI, Doug Engelbart proposed, and later implemented, a word processor with automatic word wrap, search and replace, user-definable macros, scrolling text, and commands to move, copy, and delete characters, words, or blocks of text. Stanford's TVEdit (1965) was one of the first CRT-based display editors that was widely used [Tolliver 1965]. The Hypertext Editing System [van Dam et al. 1969, p. 108] from Brown University had screen editing and formatting of arbitrary-sized strings with a lightpen in 1967 (funding from IBM). NLS demonstrated mouse-based editing in 1968 (Section 2.4.2). TECO from MIT was an early screen-editor (1967), and Emacs [Stallman 1979] developed from it in 1974 [Salus 2005]. Xerox PARC's Bravo (Section 2.4.3.2) was the first WYSIWYG editor–formatter (1974). The first commercial WYSIWYG editors were on the Star, LisaWrite, and then MacWrite. Chapter 8 discusses text editing and formatting in depth.

2.5.1.7 Spreadsheets

The spreadsheet interaction style, where the information is provided in a grid and the user can move around the grid, edit cell contents, and select various cells, rows, and columns, was initially invented by VisiCalc (1977–1978) (described in Section 2.4.4) and refined by many later systems such as Lotus 1-2-3 and Microsoft Excel.

2.5.1.8 Hypertext and the World Wide Web

The idea for *hypertext* (where documents are linked to related documents) is credited to Vannevar Bush's famous *MEMEX* idea from 1945 [Bush 1945]. Ted Nelson coined the term *hypertext* in 1965 [Nelson 1965]. Engelbart's NLS system (Section 2.4.2) in 1965 made extensive use of linking. The "NLS Journal" was one of the first online journals, and it included full linking of articles (1970) [Goldberg 1988, p. 212]. The Hypertext Editing System, jointly designed by Andy van Dam,

Ted Nelson, and two students at Brown University (funding from IBM) was distributed extensively [van Dam et al. 1969]. The University of Vermont's PROMIS (1976) was the first Hypertext system released to the user community. It was used to link patient and patient care information at the University of Vermont's Medical Center. The ZOG project (1977) from CMU was another early hypertext system and was funded by the US DOD's Office of Naval Research (ONR) and Defense Advanced Research Project Agency (DARPA, the successor to ARPA) [Robertson et al. 1977]. Ben Shneiderman's Hyperties was the first system where highlighted items in the text could be clicked on to go to other pages (1983, University of Maryland) [Koved and Shneiderman 1986]. HyperCard from Apple (1988) significantly helped to bring the idea to a wide audience (Section 2.4.9). As discussed in depth in Section 2.4.13, Tim Berners-Lee used the hypertext idea to create the Web in 1990. There have been many other hypertext systems through the years. For a more complete history of HyperText, see Nielsen [1995].

2.5.1.9 Computer-aided Design

The same 1963 IFIPS conference at which Sketchpad was presented also contained a number of Computer-Aided Design (CAD) systems, including Doug Ross's Computer-Aided Design Project at MIT in the Electronic Systems Lab [Ross and Rodriguez 1963] and Coons' work at MIT with SketchPad [Coons 1963]. Timothy Johnson's pioneering work on the interactive 3D CAD system Sketchpad III [Johnson 1963] was his 1963 MIT M.S. thesis (funded by the Air Force). The first CAD/CAM system in industry was probably General Motor's DAC-1 (c. 1963). The NEWSWHOLE system from the University of Toronto in 1975 was one of the first CAD systems applied to newspaper publishing [Tilbrook 1976, Baecker et al. 1979].

2.5.1.10 Video Games

The first graphical video game was probably *SpaceWar* by Slug Russel of MIT in 1962 for the PDP-1 [Levy 1984, p. 49] including the first computer joysticks. The early computer *Adventure* game was created by Will Crowther at BBN, and Don Woods developed this into a more sophisticated *Adventure* game at Stanford in 1966 [Levy 1984, p. 132]. Conway's game of *LIFE* was implemented on computers at MIT and Stanford in 1970. The first popular commercial game was *Pong* (about 1976).

2.5.1.11 Gesture Recognition

As explained in depth in Chapter 14, a *gesture* is a form of input where the path is important to the meaning. Sketchpad used lightpen gestures (1963). The first pen-based input device that supported gestures was the RAND graphics tablet (1964), which was funded by ARPA. Teitelman in 1964 developed the first trainable gesture

recognizer. A very early demonstration of gesture recognition was Tom Ellis' GRAIL system on the RAND tablet (1964, ARPA funded). It was quite common in lightpen-based systems to include some gesture recognition—for example, in the AMBIT/G system (1968, ARPA funded). A gesture-based text editor using proofreading symbols was developed at CMU by Michael Coleman in 1969. Gesture recognition has been used in commercial CAD systems since the 1970s and came to universal notice with the Apple Newton in 1992 (Section 3.8).

2.5.1.12 Multimedia

The FRESS project at Brown used multiple windows and integrated text and graphics (1968, funding from industry). The Interactive Graphical Documents project at Brown was the first hypermedia (as opposed to hypertext) system and used raster graphics and text, but not video (1979–1983, funded by ONR and NSF). The Diamond project at BBN (starting in 1982, DARPA funded) explored combining multimedia information (text, spreadsheets, graphics, speech). The Movie Manual at the Architecture Machine Group (MIT) was one of the first to demonstrate mixed video and computer graphics in 1983 (DARPA funded).

2.5.1.13 3D

The first 3D system (see Chapter 12) was probably Timothy Johnson's 3D CAD system (1963, funded by the Air Force) [Johnson 1963]. The *Lincoln Wand* by Larry Roberts was an ultrasonic 3D location sensing system developed at Lincoln Labs (1966, ARPA funded). That system also had the first interactive 3D hidden line elimination. An early use was for molecular modeling [Levinthal 1966]. The late 1960s and early 1970s saw the flowering of 3D raster graphics research at the University of Utah with Dave Evans, Ivan Sutherland, G.W. Romney, Henri Gouraud, Bui-Tuong Phong, and G.S. Watkins, much of it government funded. Also, the military–industrial flight simulation work of the 1960s and 1970s led the way to making 3D interfaces operate in real time on commercial systems from GE, Evans & Sutherland, and Singer/Link (funded by NASA, Navy, etc.). Another important center of early research on 3D interfaces was Fred Brooks' lab at UNC (e.g., Brooks [1977]).

2.5.1.14 Virtual Reality and Augmented Reality

The original work on Virtual Reality (see Chapter 12) was performed by Ivan Sutherland when he was at Harvard (1965–1968, funding by Air Force, CIA, and Bell Labs). Very important early work was done by Tom Furness when he was at Wright-Patterson AFB. Myron Krueger's early work on VideoPlace showed interfaces that

incorporated and interpreted a video of the user done at the University of Connecticut. It demonstrated, among other things, shape control by multiple fingers and a virtual "critter" that interacted with the user [Krueger et al. 1985], and was very influential. Fred Brooks' and Henry Fuch's groups at UNC did much early research, including the study of force feedback (1971, funding from the US Atomic Energy Commission and NSF). Much of the early research on head-mounted displays (Figures 12.1 and 12.2) and on the Data Glove (Section 12.4.3) was supported by NASA.

2.5.1.15 Computer-supported Cooperative Work

Doug Engelbart's 1968 demonstration of NLS (Section 2.4.2) included the remote participation of multiple people at various sites (funding from ARPA, NASA, and Rome ADC). Licklider and Taylor [1968] predicted online interactive communities and speculated about the problem of access being limited to the privileged. Electronic mail, still the most widespread multi-user software, was enabled by the ARPAnet, which became operational in 1969 (and evolved into today's Internet), and by the Ethernet from Xerox PARC in 1973. An early computer conferencing system was Turoff's EIES system at the New Jersey Institute of Technology (1975).

2.5.1.16 Natural Language and Speech

The fundamental research for speech and natural language understanding and generation has been performed at CMU, MIT, SRI, BBN, IBM, AT&T Bell Labs, and BellCore, much of it government funded. See, for example, Reddy [1990] for a survey of the early work and Chapter 18 for a discussion of intelligent interfaces in general.

2.5.2 Implementation Techniques

The area of user interface software tools has been quite active, and many companies have sold tools that help create interactive technologies. Most of today's applications are implemented using various forms of software tools. For a more complete survey and discussion of early UI tools, see Myers [1995]. Chapter 20 discusses ways to implement interaction techniques.

2.5.2.1 User Interface Management Systems and Toolkits

The term *User Interface Management System* (UIMS) was an old name for software tools that tried to help programmers create user interfaces. The term was coined by David Kasik at Boeing as an analogy to database management systems (1982) [Kasik 1982]. The term is no longer used. A more current name is "Toolkits," which are software libraries that help to create user interfaces. The first UIMS was William

Newman's Reaction Handler [Newman 1968b] created at Imperial College, London (1966–1967 with SRC funding). Most of the early work was done at universities (University of Toronto with Canadian government funding, George Washington University with NASA, NSF, DOE, and NBS funding, Brigham Young University with industrial funding, etc.). Early window managers (see Chapter 11) such as Smalltalk (1974) and Interlisp, both from Xerox PARC, came with a few widgets such as popup menus and scrollbars. The Xerox Star (1981) was the first commercial system to have a large collection of widgets (Section 2.4.3.4). The Apple Macintosh (1984) was the first to actively promote its toolkit for use by other developers to enforce a consistent interface. An early C++ toolkit was InterViews [Linton et al. 1989], developed at Stanford (1988, industrial funding). Much later research on user interface tools was also done at universities—for example, my work on Garnet (1988) [Myers et al. 1990] and Amulet (1994) [Myers et al. 1997a] at CMU (DARPA funded) and Hudson's work on subArctic at Georgia Tech (1996, funding by Intel and NSF) [Hudson et al. 2005].

2.5.2.2 Interface Builders

Interface builders are interactive tools that allow interfaces that are composed of widgets such as buttons, menus, and scrollbars to be placed using a mouse (see Section 20.8). The Steamer project [Stevens et al. 1983] at BBN (1979–1985; ONR funding) demonstrated many of the ideas later incorporated into interface builders and was probably the first object-oriented graphics system. Trillium [Henderson Jr. 1986] was developed at Xerox PARC in 1981. Another early interface builder was the MenuLay system [Buxton et al. 1983b] developed by Bill Buxton at the University of Toronto (1983, funded by the Canadian government). The Macintosh (1984) included a "Resource Editor" that allowed widgets to be placed and edited. Jean-Marie Hullot created *SOS Interface*, which was the first modern interface builder, in Lisp for the Macintosh while working at INRIA (1984, funded by the French government) (see Section 20.8). Hullot built this into a commercial product in 1986 and then went to work for NeXT and created the NeXT Interface Builder (1988), which popularized this type of tool.

2.5.2.3 Component Architectures

The idea of creating interfaces by connecting separately written components was first demonstrated in the Andrew project [Palay et al. 1988] by CMU's Information Technology Center (1983, funded by IBM). This architecture was then widely popularized by Microsoft's OLE, Apple and IBM's OpenDoc, and the Java "Beans" architectures, and now has morphed into web services.

2.6 Chapter Conclusion

This chapter has summarized many of the important stages of the development of computers, especially as it relates to interaction techniques. This short survey is necessarily incomplete and somewhat idiosyncratic to what I thought was important and interesting. The next chapter does the same thing for the advances that led to today's smartphones, tablets, watches, and other mobile devices.

CHAPTER 3

History of Mobile Devices

3.1 Introduction

The previous chapter focused on "regular" computers, nowadays often called *desktop* computers, but most new interaction techniques are being invented for devices that can be carried around. Figures 3.1 and 3.2 show why—Windows computers dominated the 1990s, but since 2010 mobile devices have been the key computing platform the world uses by an increasingly wide margin. This chapter gives a brief history of important mobile platforms. But first, a few definitions.

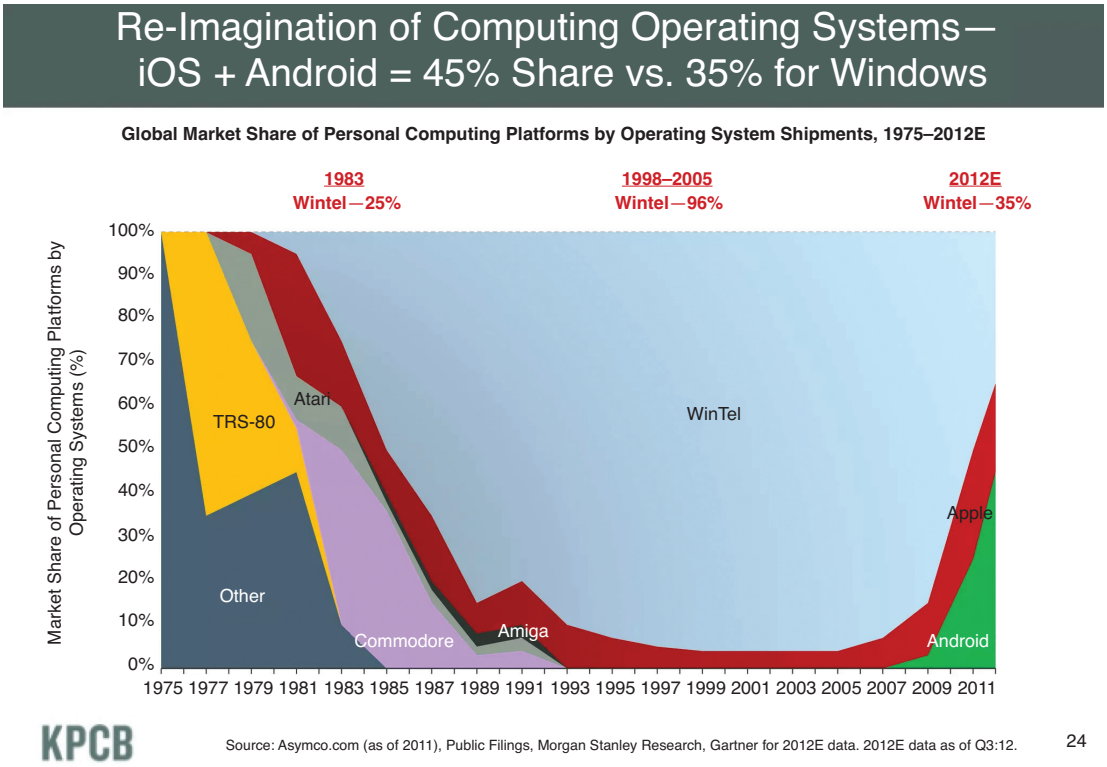


Figure 3.1 The rise and fall of Windows computers. Source: Slide 24 of http://www.slideshare.net/kleinerperkins/2012-kpcb-internet-trends-yearend-update.

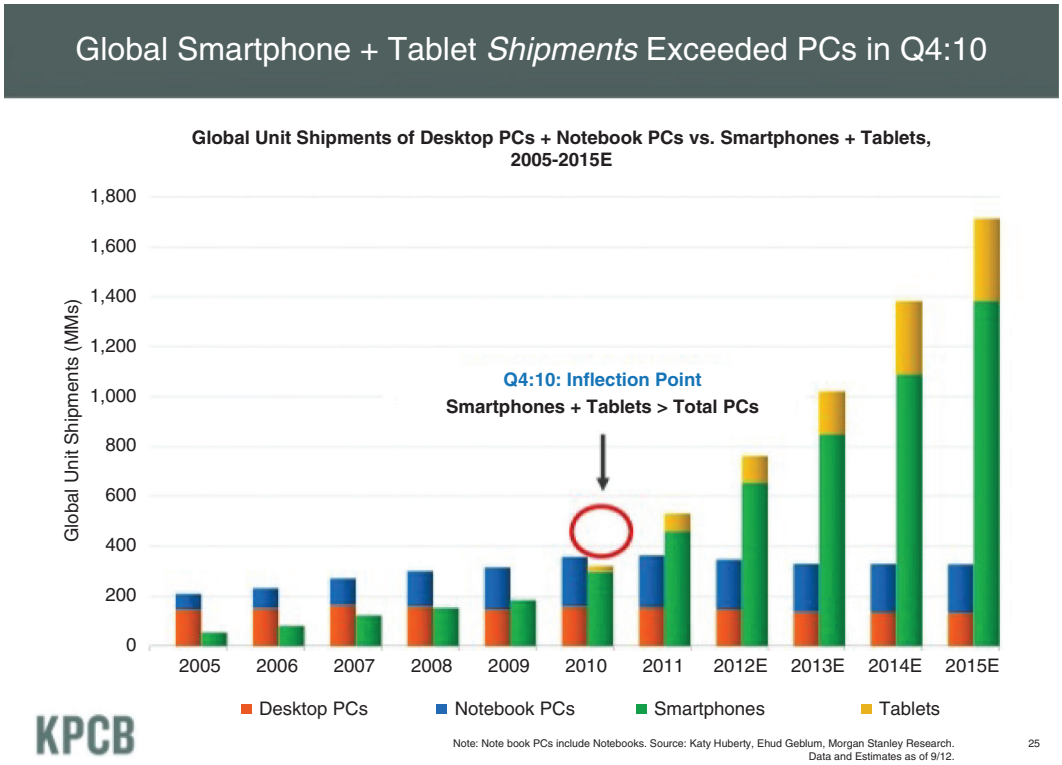


Figure 3.2 Most computing platforms shipped have been mobile since 2010. Source: Slide 25 of http://www.slideshare.net/kleinerperkins/2012-kpcb-internet-trends-yearend-update.

3.2 Definitions

3.2.1 Mobile Devices and Handheld Devices

Wikipedia defines a *mobile device* as "a computing device small enough to hold and operate in the hand." This definition seems fine to me. Notice that *laptop* computers do *not* count as mobile devices under this definition since they are small enough to be *carried* in a hand, but they cannot really be *operated* without putting them on the table. Somewhat in the middle are devices like the HP Jornada 720 (Figure 3.3), which are like mini-laptops and were called *handheld PCs* at the time. I will count these as mobile devices since they could easily be held in one hand while operated with the other. Their tiny keyboards were not really designed to support touch-typing, as one would do with a laptop or desktop computer, which requires both hands.

Wikipedia equates *handheld computers* or *handheld devices* (or just *handhelds*) as being the same as mobile devices. However, I will use *handheld PC* for the clamshell devices like Figure 3.3, which are just one type of handheld device.

Figure 3.3 HP Jornada 720 handheld PC, from about 2000. The size is 7.40in by 3.70in by 1.30in. Source: Matthew850, public domain, https://commons.wikimedia.org/wiki/File:HP_Jornada_720.JPG.

The term *portable device*, on the other hand, can be used for any device that is designed to be carried around and would include laptops and other devices that are sufficiently light to be easily moved. A portable device does *not* need to be *operated* while being carried.

3.2.2 Touchscreens and Touchpads

A *touchscreen* is a display screen that the user touches as input. Note that devices that are *not* mobile can still have touchscreens, like touch-sensitive wall-size displays. Also, it is possible for a mobile device's screen to *not* be a touchscreen. For example, old mobile phones were mobile but did not have touchscreens. In contrast, a *touchpad* is an input device that is sensitive to touch but is not a display screen. For example, many laptops have touchpads next to the keyboard (see Figure 3.4).

3.2.3 Palm-size Devices, Tablets, and Wearables

There are many different kinds of mobile devices, with the most popular separated into three main classes: palm-size devices, tablets, and wearables. A *palm-size device* is a mobile device designed to fit into the palm of one's hand.

In contrast, a *tablet device* is a mobile device that is too big to fit in a person's palm but small enough to be held on a person's forearm. Typically, tablets have a touchscreen and no physical keyboard.

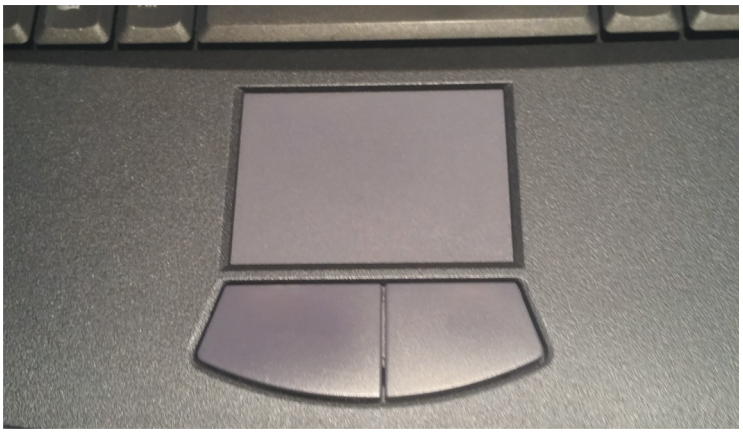


Figure 3.4 AV touchpad on a 2000s-era Sony Vaio laptop, an early example of a modern laptop touchpad. Source: By CinemaLover24680—Self-photographed, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=63034244.

Finally, *wearable devices* are smaller mobile devices that are designed to be worn on a person's body. Typical examples of wearable devices include smart wrist watches and computerized eyeglasses.

3.2.4 Pen Computing

An early name for mobile devices was *pen computing* since it was assumed that most of the interaction on a mobile device would use a *stylus*, which is a pointy stick that is shaped much like a pen. Figure 3.5 shows examples of various kinds of styluses.

Figure 3.5 Several styluses: (L to R) PalmPilot Professional, Fossil Wrist PDA, Nokia 770, Audiovox XV6600, HP Jornada 520, Sharp Zaurus 5500, and Fujitsu Lifebook P-1032. Source: By Craig Spurrier—Own work, CC BY 2.5, https://commons.wikimedia.org/w/index.php?curid=2000599.

3.2.5 Gestures

A *gesture* (as an interaction technique) is a form of input where the *path* is important to the meaning (as defined in Section 2.5.1.11 and fully explained in Chapter 14). An example of a gesture is drawing a circle on a screen to indicate that a circle object is desired, as opposed to drawing a box to indicate a rectangle should be drawn. *Flicking* with a finger on a phone is also a gesture since the origin and direction of movement of the finger is relevant to know which operation will happen. And waving your hands in front of a gaming console to control a virtual sword is also a gesture. So, *gesture recognition* is possible with devices that are not mobile, but it has been mostly used on mobile devices.

3.2.6 Handwriting Recognition and Hand Printing Recognition

A particular kind of gesture recognition is *handwriting recognition*, which is where users can write letters and words as a way to input text to a computer (see Section 8.3.1). Typically, the user would use a stylus on a touchscreen so that the writing can be as close as possible to the way the user would normally handwrite. A subclass is *hand printing recognition*, which requires that each character be separately printed, making it easier for the computer to tell the characters apart—as opposed to cursive writing where the letter separation must be done by the recognition algorithm.

3.2.7 Personal Digital Assistants

The term PDA was coined by Apple in approximately 1993 to describe their *Newton* product (see Section 3.8). This term was generalized to mean any mobile device that serves as a personal information manager, supporting such functions as a contact manager (phonebook) and calendar. The original PDA devices were not mobile phones.

3.2.8 Mobile Phones, Feature Phones, and Smartphones

Mobile phones, also called *cell phones*, are devices with which users can make and receive phone calls over a radio frequency (RF) link. The early mobile phones only supported calling, but it quickly became clear that it would be useful to provide other features. This led to what are now called *feature phones*, which typically supported some features besides telephony such as having a phonebook for a user's contacts. Usually, feature phones had a regular telephone 12-button keypad and a small screen that was not a touchscreen. Next came *smartphones*, which combine all PDA functions with telephony and typically have a full keyboard, either as a

Figure 3.6 HP-65 calculator—first programmable calculator. Source: Uploaded by teclasorg, CC-BY-SA-2.0, https://commons.wikimedia.org/wiki/File:HP_65.jpg.

small physical keyboard or an on-screen keyboard (see Section 3.10 below and also Chapter 8 about text entry).

3.3 Calculators

Probably the first mobile devices were *programmable calculators*. I remember getting one of the first calculators from Texas Instruments (TI) in the early 1970s. The first calculator that was programmable by the user was the Hewlett Packard (HP) 65 from 1974 (see Figure 3.6), and the first graphing calculator was the Casio FX-7000G released in 1985, according to Wikipedia. HP and TI continue to fight over the calculator and programmable calculator market through today. The interaction techniques on calculators remained very simple—just tapping on small physical keys, where often one or more of the keys caused the other keys to change modes and enter different characters. However, the way the user entered the calculations was quite different, with the TI using regular *infix* notation with the operator in the middle (3 + 4) while the HP used *Reverse Polish Notation* with the operator last (3, 4, +), as explained in Section 10.2.

3.4 Ubiquitous Computing

Xerox PARC had two tremendously influential periods. The first was the 1970s with the invention of many of the interaction techniques for regular computers (Section 2.4.3). The second was the late 1980s and early 1990s when Mark Weiser at PARC coined the term *ubiquitous computing* (sometimes abbreviated *Ubicomp*) [Weiser 1991, 1993] and demonstrated many properties of mobile devices. Ubiquitous computing refers to computation that is available anytime and anywhere, in a manner where "[the devices] weave themselves into the fabric of everyday life

Figure 3.7 Xerox PARC's tabs, pads, and board-size devices. Source: Weiser [1993].

until they are indistinguishable from it" and become "invisible" [Weiser 1991]. Weiser led a group that investigated computation in three form factors: tabs, pads, and board size. Tabs were palm-size mobile devices, pads were tablets, and board-size devices aimed to be the size of a whiteboard where multiple people could collaborate (see Figure 3.7).

The PARC tab, developed around 1989, used a low-speed wireless network based on infrared (IR) signals, like what remote controls use (Wi-Fi would not be generally available until years later, in the 1990s, see Section 3.12). It innovated the important concept that the small, portable devices would be continuously connected to the network, so expensive computations could be off-loaded [Want et al. 1995]. Another important innovation on the tab was a text entry technique called *unistrokes* where each character is entered with a single stroke, drawn on top of each other, to provide quick and highly reliable text entry [Goldberg and Richardson 1993] (see Section 8.3.2). Xerox commercialized a version of the board-size devices but left the commercial versions of the tab and the tablet to others.

3.5 Pen Operating Systems

The Pencept company provided the PenPad tablets in the early 1980s, with products launched in 1982. It provided hand printing recognition and other gestures for input [Ward and Blesser 1985].

One of the first commercial attempts at a self-contained pen-based computer was the PenPoint operating system from Go Corporation. The company was

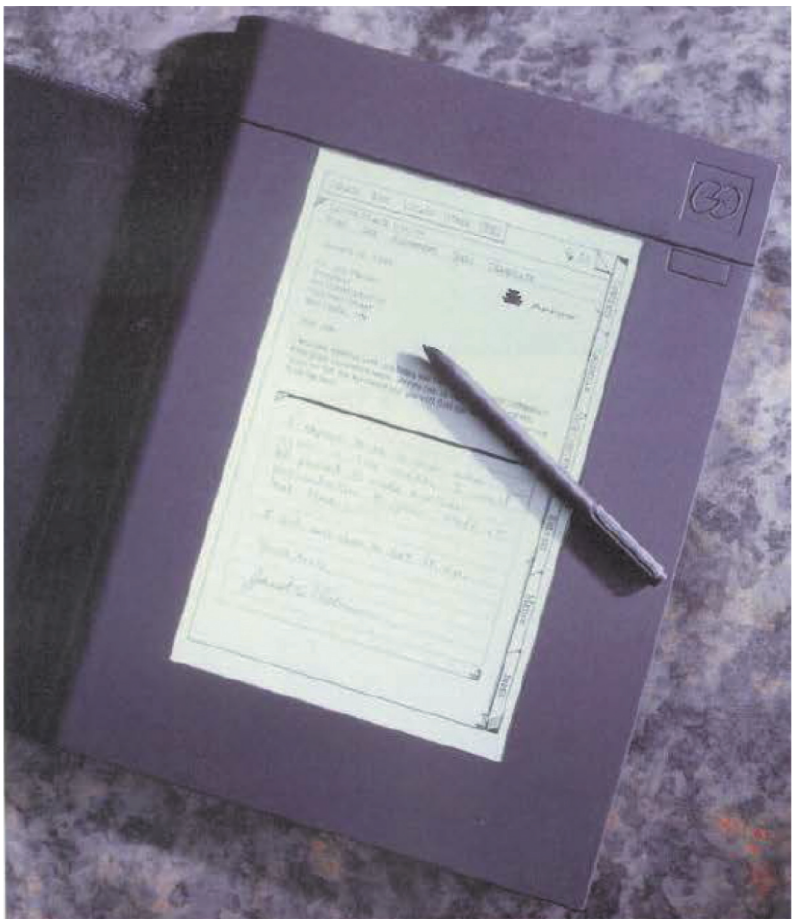
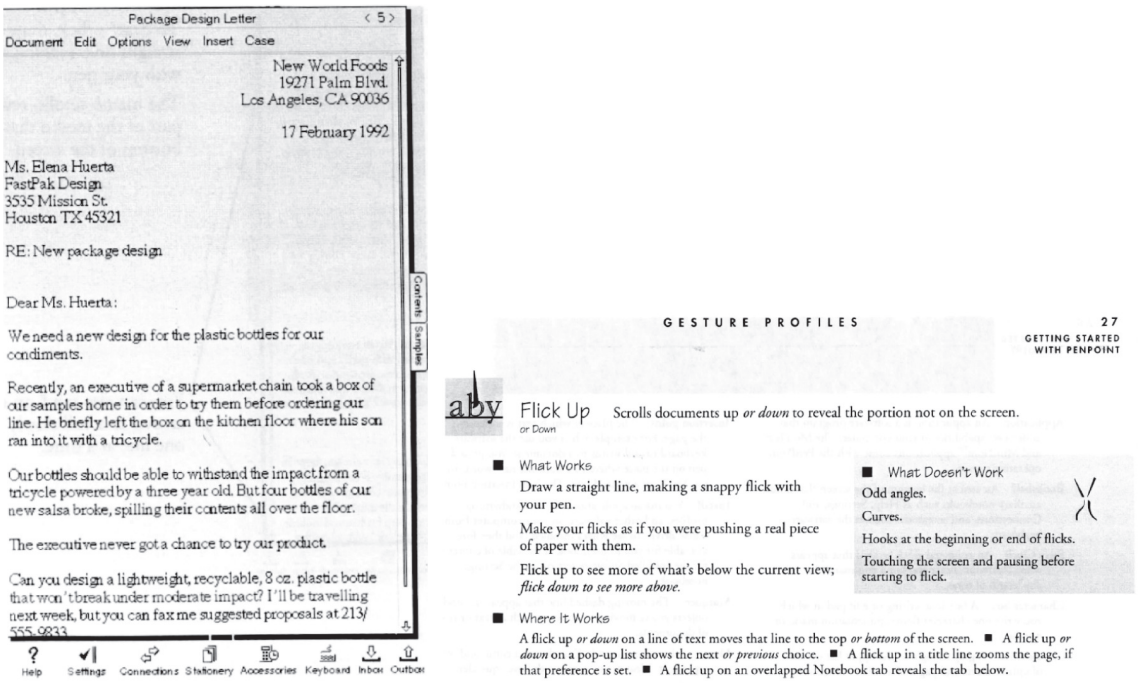


Figure 3.8 Reference hardware for Go Corporation's PenPoint system. Source: Courtesy DigiBarn Computer Museum.

founded in 1987, with the first product released in 1991. Their concept was that they would create the software and some of the main applications, but others would create the hardware. Figure 3.8 shows the reference hardware for developers, but the intention was that the commercial versions would be sleeker and lighter. Eventually, versions of the hardware were marketed by NCR, IBM, and EO. One of the founders of Go was Robert Carr from Xerox PARC, who was one of the designers of the Alto hardware.

Most interesting about PenPoint was the look and feel of the UI. It was styled to look like a tabbed notebook (see the right side of Figure 3.9(a)), and tapping with the stylus on a tab would bring it to the front. Many conventional interaction techniques were supported, such as tapping on buttons and menus (see the top of Figure 3.9(a)) and press-and-hold to start moving or selecting. In addition, it had a



(a) (b)

Figure 3.9 (a) A page from the PenPoint user's guide showing tabs, icons, a scrollbar, and menus. (b) The "Flick" gesture for scrolling. Source: Go Corporation [1992, pp. 12, 27].

full set of gestural commands including flick-to-scroll (see Figure 3.9(b)), circling text to select it, crossing out to delete items, and hand printing (or an on-screen keyboard) for text entry and editing. PenPoint also had clickable tables of contents and indexes that could serve as a form of hyperlinks, and it supported instant-on and off, so it could be picked up and used without waiting for a boot sequence.

3.6 GRiDPad

Another important early company in the pen computing space was GRiDPad, founded around 1989 by Jeff Hawkins (who would later help found Palm Corporation, Section 3.13). The GRiDPad hardware was fairly large, heavy, and clunky (like the PenPoint hardware), weighing about 5lb. It had a 20MHz processor with 20MB of RAM and a 40MB, 60MB, 80MB, or 120MB hard drive. It had a 10-inch diagonal backlit VGA display with 32 gray scales. There was a built-in PCMCIA card slot, an internal fax/modem card, a floppy drive port, and a standard keyboard port. Operating time was about three hours on a NiCad battery pack [Flynn 2005].

(a) (b)

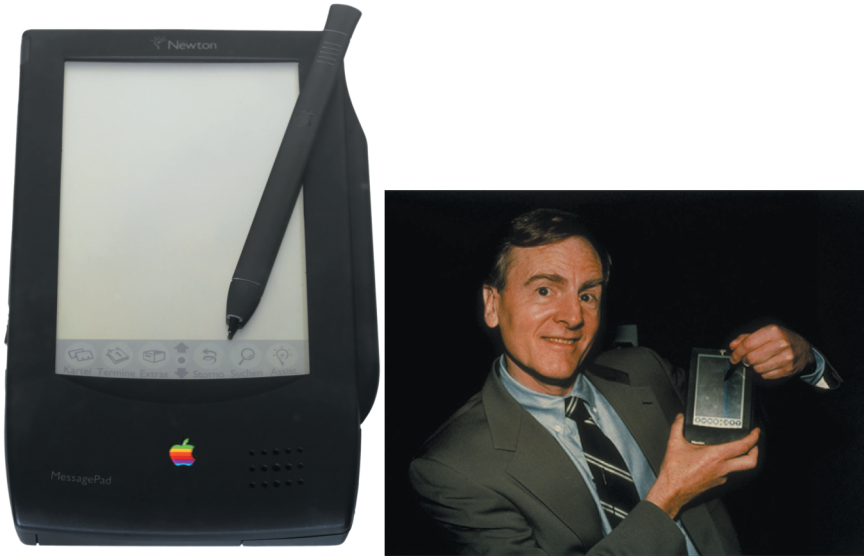
Figure 3.10 Microsoft PenWindows. Sources: (a) Image courtesy of Charles Headey, https://retrocosm.net/2012/01/. (b) Image courtesy of Betaarchive.com, https://www.betaarchive.com/imageupload/1298947809.or.94950.png.

3.7 Microsoft Pen Windows

Microsoft came out with a version of Windows 3.1 for pen computing, around 1991, called Pen Windows or Windows for Pen (Figure 3.10). This was to run on various kinds of hardware where the primary input would be a pen. However, most of the interaction techniques were the same as for regular Windows, requiring users to tap on tiny menu items and icons, which could be tricky with a pen. Text entry was supported from an onscreen keyboard but also from either hand printing or handwriting recognition, which was not particularly accurate. Later, pen versions of Windows NT, Windows 95, and other operating systems were also released.

3.8 Apple Newton

Steve Jobs left Apple in 1985 and rejoined the company in 1997 (meanwhile, he founded NeXT, see Section 2.4.11) [Isaacson 2011]. For the first part of the time that Jobs was gone, John Sculley III was in charge of Apple, and a key new Apple product from that era was the Newton MessagePad, which was started in 1987 and released in 1993. As mentioned above, Apple coined the term PDA to describe the Newton, and it supported the standard personal information tasks like contacts and calendar. As it was originally shipped, it did not have a wireless connection, and it did not easily sync with a regular computer, which limited its usefulness. The Newton had an interesting operating system and its own custom programming language called NewtonScript, which used a different kind of object-oriented



(a) (b)

Figure 3.11 Apple Newton (a) MessagePad 100. (b) Demonstration by John Sculley III. Sources: (a) Photograph by Rama, Wikimedia Commons, Cc-by-sa-2.0-fr CeCILL (http://www.cecill.info/licences/Licence_CeCILL_V2-en.html) https://commons.wikimedia.org/wiki/File%3AApple_Newton-IMG_0454-cropped.jpg. (b) Getty Images, http://techland.time.com/2012/06/01/newton-reconsidered/.

programming model from C++ and Java called a *prototype-instance object model* (see Section 19.2).

The device was expensive and the hardware (see Figure 3.11) was somewhat large and heavy, but the best-known problem with the Newton was the text entry

Figure 3.12 Doonesbury comic strip from August 26, 1993 about the Apple Newton. Source: DOONESBURY © 1993 G. B. Trudeau. Reprinted with permission of ANDREWS MCMEEL SYNDICATION. All rights reserved.

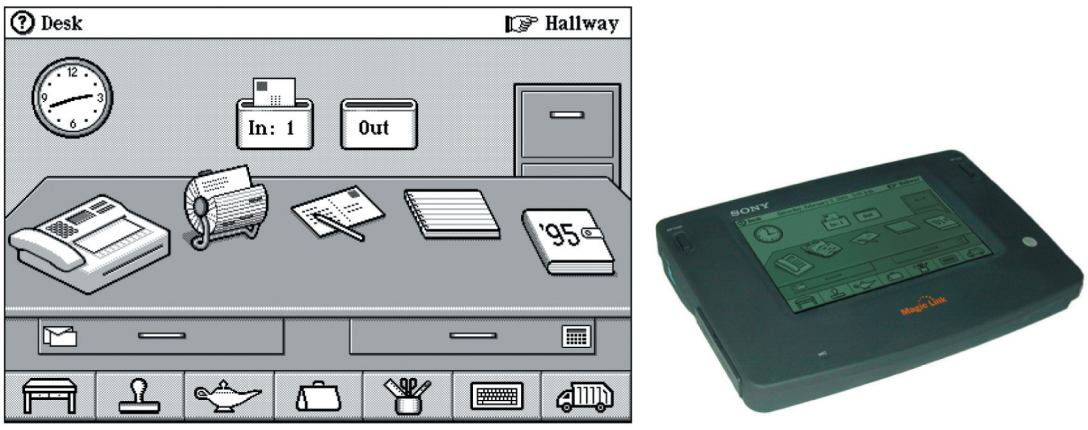
technique, which focused on handwriting recognition—this, unfortunately, was not very good. In fact, the Newton was panned for an entire week in August 1993 by the Doonesbury comic strip (see Figure 3.12). The device went through a number of versions and was eventually taken off the market in 1998.

3.9 General Magic's "Magic Cap" OS

A little known but cool mobile device was General Magic's Magic Cap operating system that ran on Sony MagicLink hardware, released in 1994, and partially designed by Bill Atkinson (see Section 28.4). The General Magic company, like Go Corporation (Section 3.5), tried to make a business of supplying a special operating system for mobile devices to run on other companies' hardware. In this case, Magic Cap focused on office tasks, especially email but also PDA features like an address book. The device used a special AT&T wireless data network, which was very slow. The user model was a three-dimensional (3D) representation of an office, with icons for the various activities (see Figure 3.13).

3.10 Early Wireless Phones

There were no interesting interaction techniques for the original mobile phones, but feature phones did introduce new ways of interacting, especially for text entry. In 1993, Nokia introduced a multi-line display screen and up/down navigation buttons to support interactions. Soon, up/down and four-way navigation was supported by most phones, as more features became commonplace. Text entry



(a) (b)

Figure 3.13 (a) General Magic's "Magic Cap" running on (b) Sony MagicLink hardware. Sources: (a) By Source, Fair use, https://en.wikipedia.org/w/index.php?curid=9351516. (b) By Josh Carter, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=52495067.

Figure 3.14 Motorola Razr with keypad. Source: Peterwhy at English Wikipedia, Public domain, via Wikimedia Commons, https://commons.wikimedia.org/wiki/File:RAZR_V3i_opened.JPG.

involved tapping on the number keys multiple times in what is called *multi-tap* mode—for example, hitting the "2" key three times to get a "C" character or else the trademarked *T9* mode that would predict which word was likely wanted from one tap on each number key (see detailed discussion in Section 8.3). From a hardware standpoint, during this period phones got thinner and thinner, culminating in the super-thin Motorola Razr starting in 2003 (see Figure 3.14).

When the World Wide Web became popular in the 1990s, it became clear that it would be a good idea to access it from cell phones, but the wireless networks were slow (see Section 3.12) and the screens on the phones were tiny. Therefore, vendors supported a standard called the *wireless access protocol* (WAP), released in 1997, which provided simplified pages of information. Unfortunately, WAP had terrible usability [Nielsen 2000], and few companies provided WAP versions of their web pages, so WAP never gained widespread popularity.

3.11 Early Smartphones

As defined above, a *smartphone* is a device with both telephone and PDA features, typically with a full keyboard (as opposed to a telephone 12-button keypad). Possibly, the earliest commercial smartphone was the IBM Simon (see Figure 3.15),

Figure 3.15 IBM Simon from 1994. Source: By Bcos47 Public Domain, https://commons.wikimedia.org/w/index.php?curid=20083707.

which saw limited release by BellSouth starting in 1994. Two years later, in 1996, Nokia released the 9000 Communicator with a unique design—when it was closed, it looked like a regular (if fat and heavy) mobile phone with a 12-key keypad (Figure 3.16(a)). When opened, it looked like a clamshell PDA—a handheld PC (Section 3.2.1) with a horizontal touch display and a full keyboard (Figure 3.16(b)). Although both of these devices had most of the features we would expect today,

(a) (b)

Figure 3.16 Nokia 9110 Communicator: (a) closed and (b) open. Sources: Both by Oldmobil—Self-photographed, CC BY-SA 3.0: (a) https://commons.wikimedia.org/w/index.php?curid=7861764. (b) https://commons.wikimedia.org/w/index.php?curid=7861738.

they were too much ahead of their time and the processor and phone networks were too slow (as discussed below in Section 3.12, its data was only 2G).

3.12 Wireless Networks

With the rise in popularity of mobile devices of all kinds, it was apparent to all that high-speed wireless connectivity for data transfer was necessary. The most obvious way to do this was using the mobile phone networks. These went through a number of *generations*, typically abbreviated *1G*—1st generation, *2G*, and so on. Table 3.1 shows the various data speeds of the generations. Useful access to the Internet required at least 3G speeds.

Separately, small area wireless networks were being developed. The IR network used by PARC (Section 3.4) proved impractical since it required line-of-sight and was blocked by people and furniture. Therefore, RF techniques dominated. What we now call Wi-Fi started around 1988 and was called IEEE 802.11, which is the name of the standard that defines it. This is actually a family of standards, like 802.11a, 802.11b, and so on. Carnegie Mellon University (CMU) was one of the first places to use it and became the first fully wireless campus in 1997 in a project called Wireless Andrew [Hills 1999] ("Andrew" is the first name of both of CMU's founders—Andrew Carnegie and Andrew Mellon). At the time, the network was also called WaveLAN, Lucent's trademark name for the early implementation. The cards implementing this protocol were mainly used with laptops because they were too big and required too much power for most mobile devices (see Figure 3.17). The name Wi-Fi was trademarked in 1999 to apply to this whole class of wireless technologies, and eventually the implementations became small enough for mobile devices.

Meanwhile, starting in 1994, phone companies, led by the Swedish company Ericsson, were developing a very low-power radio standard to connect mobile

Table 3.1 **Four mobile phone generations**

	Dates	Data speeds	Standards	Features
1G	1980s	None	AMPS, TACS	Analog transmission
2G	1992	40 kbit/s up to 236.8 kbit/s	GSM, CDMA, EDGE, GPRS	Digital for data, analog for voice; SMS
3G	2001	200 kbit/s up to 384 kbit/s	UTMS, CDMA2000, HSPDA, EVDO	Packet switching for data
4G	2008	100 Mbit/s, up to 1 Gbit/s	LTE Advanced, IEEE 802.16 (WiMax)	All digital
5G	2019	20 Gbit/s peak	IEEE 802.11ad	Approaches Wi-Fi speeds

Figure 3.17 Lucent WaveLAN PCMCIA format card for laptops.

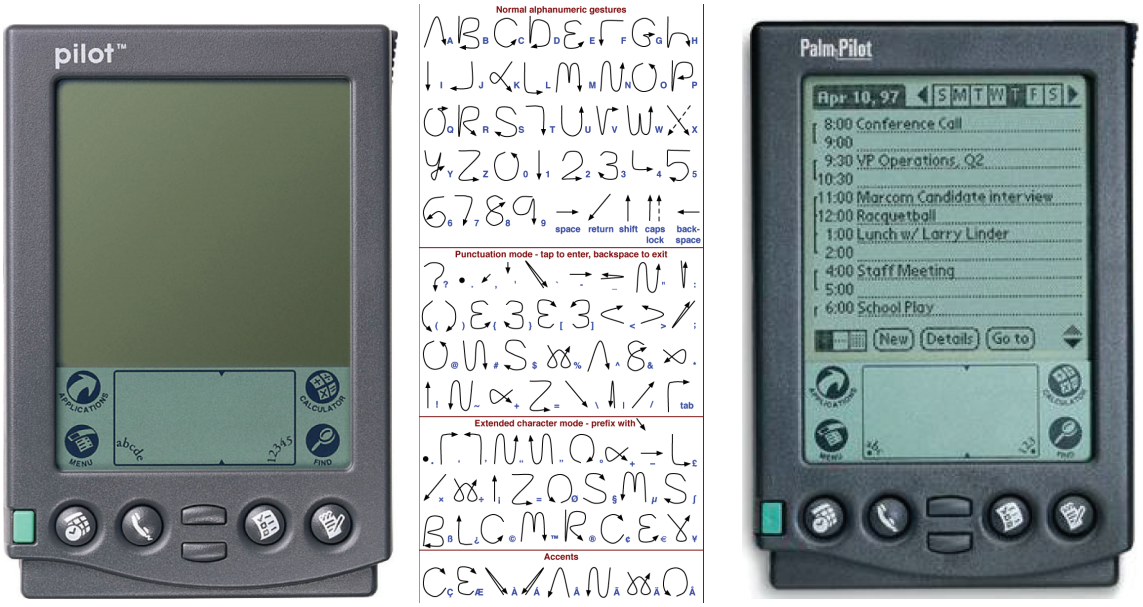
phones with headphones. The protocol was named Bluetooth in 1997 after a tenth-century Danish king, although the official standard name is IEEE 802.15. This protocol is short-range by design and can only connect two devices. It is now used for a variety of one-to-one communication needs like a computer to a mouse or a phone to a car's speakers.

3.13 Palm

The Palm (1996) was a breakthrough product in mobile devices. The company was founded by Jeff Hawkins, who had previously worked on the GRiDPad (Section 3.6) in 1992. Originally, the company, called Palm Computing, was only going to produce software, but the founders decided they could produce better hardware as well. In 1995, before the first version was released, Palm Computing was acquired by US Robotics, which was then acquired by 3Com in 1997. Some members of the original company created a competing company in 1998 called Handspring, which re-merged with the original company to again form Palm Computing in 2000. That company was purchased by HP in 2010, but by then the product was essentially dead.

The first released product came out in 1996 and was called the Pilot (see Figure 3.18(a)), but this name was changed due to a trademark infringement lawsuit from the Pilot pen company (since the Palm includes a stylus), and later versions were just called *Palm* devices.

The designers of the Palm did a significant amount of user testing on their device that contributed to its widespread acceptance. The case was designed to a size that would fit into a shirt pocket and was substantially smaller and lighter than previous pen computers. Hawkins said he carried a block of wood, the size and weight of the potential Pilot, in his pocket for a week to make sure it was



(a) (b) (c)

Figure 3.18 (a) Original Palm Pilot from about 1996. (b) The Graffiti alphabet for Palm. (c) Palm Pilot. Sources: (a) Screen capture, May 2018. (b) IMeowbot, CC-BY-SA, https://commons.wikimedia.org/wiki/File:Palm_Graffiti_gestures.png. (c) Contributor, fair use, https://techcrunch.com/2009/02/12/i-remember-when/.

comfortable. The software was also user tested, with prototypes created using Apple's HyperCard program.

The designers of the Palm articulated a number of important design principles [Bergman and Haitani 2000] that helped make the Palm one of the easiest devices of its time to use. A key insight was that the usage context of a mobile device would be quite different than a regular computer—in that people would want to do a task quickly and immediately. For example, a user might take the Palm out of a pocket and want to check the next appointment. It would not be acceptable to have to wait for the device to boot up or to require a lot of menu choices. Therefore, the designers identified a small number of tasks that most users would want and assigned these to four application buttons (see Figure 3.18(a)). Similarly, there were hardware or on-screen buttons for the user's main tasks within an application, such as scrolling or creating a new calendar entry. Interestingly, this often made the UI appear non-symmetric and inconsistent with the way actions would be on a regular computer. For example, the calendar application (Figure 3.18(c)) had a prominent "New" button, but to delete a calendar entry required going through a dialog box or menu. The designers had identified that users often wanted to create new

entries easily and quickly, but users rarely deleted any since they just disappeared into the past on their own. Similarly with contacts and many other items on the Palm.

Another observation is that applications should open to the most useful place for users, so the calendar always opens showing *now*. In contrast, desktop computer applications typically always open on a landing or blank page or where users were last time. Also, this contrasts with the design philosophy of the Microsoft competitor to the Palm (see Section 3.14), which tried to make it so that if one knows a PC, the mobile devices would work in a familiar way, even if this was slower. However, since the usage models of a PC (with long usage of applications like mail and document editors) differ so drastically from a mobile device (with quick and short usage of apps like a calendar or contacts), it does not make sense to make them work the same way.

The text entry on the Palm was designed to be more reliable than Newton's handwriting recognition. Jeff Hawkins had seen Xerox *unistrokes* (Section 3.4) and used some of the concepts, including that each character should be entered with a single stroke and that the characters should be written directly on top of each other. Xerox sued Palm since unistrokes was patented [Goldberg 1993]. After years of trials and appeals, Palm ended up paying $22.5 million to Xerox to settle all the lawsuits [Smith 2006]. The Palm version, called Graffiti, changed the stroke for each letter to be more memorable—using strokes that are similar to the upper- or lowercase way to draw each letter (see Figure 3.18(b)). The Palm input area is below the screen (see Figure 3.18(a)) and is divided into a letter area on the left and a number area on the right since numbers and letters can share the same gesture (e.g., "I" and "1" are both a single downward stroke).

There was also a wristwatch made by the Fossil company that ran the Palm UI. This was announced in 2002 but did not start shipping until 2003, and was on sale until about 2005. It has a tiny 160×160-pixel screen that, like the Palm, was designed to be used with a tiny stylus (see Figure 3.19), which was integrated into the band. A key problem was that it used a rechargeable battery that did not last even all day. It was also relatively heavy and expensive for the time, costing $250.

The Palm UI was also integrated with a variety of mobile phones, starting around 2001. For example, the Kyocera QCP-6035 (Figure 3.20(a)) came out in 2001 and featured a flip phone design with physical phone buttons on the front or the regular Palm UI on the inside when it was flipped open. Later, the Handspring company created the Palm Treo line that used a tiny *thumb keyboard* instead of Graffiti for text entry (Figure 3.20(b)), starting around 2002.

(a) (b)

Figure 3.19 (a) Palm watch by Fossil. (b) The tiny stylus could be stored in the band.

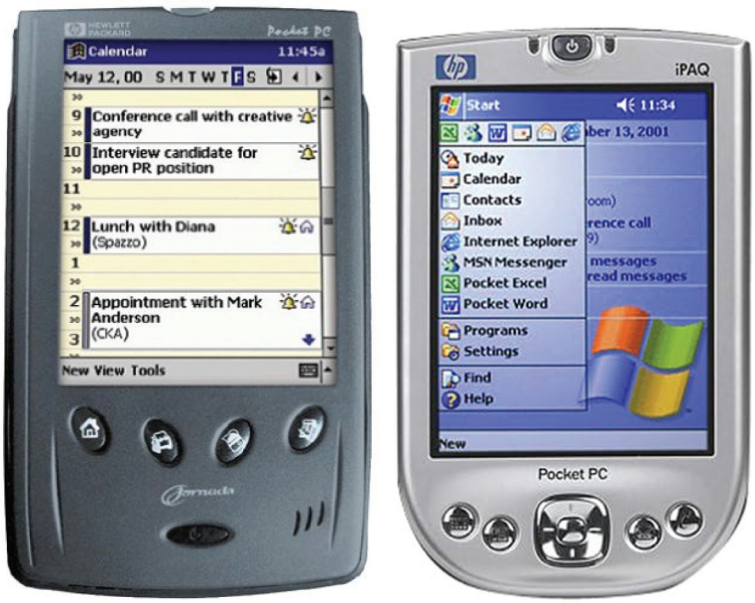
(a) (b)

Figure 3.20 (a) Kyocera QCP-6035 Palm flip phone. (b) A color Palm Treo. Sources: (a) KeithTyler at English Wikipedia, Public domain, via Wikimedia Commons, https://upload.wikimedia.org/wikipedia/commons/4/4a/Kyocera6035.jpg. (b) Giaccai, CC BY-SA 4.0 via Wikimedia Commons, https://commons.wikimedia.org/wiki/File:PALM_Treo_650.jpg.

3.14 Windows CE, PocketPC, etc.

Meanwhile, Microsoft did not want to be left out of the pen computing market. Their earlier approach with Pen Windows (Section 3.7) was not successful, so they tried a different approach of creating a new version of the Windows operating system. A key requirement was that it be instant-on and not require a long boot sequence like regular Windows. It also needed to support the kinds of interaction techniques expected for mobile devices, as described above.

The first version was called Windows CE (for "compact edition") and was released in 1996 (the same year as the first Palm Pilot). Over its lifetime, this operating system had a variety of other names as well, including Windows Palm PC (which Microsoft was required to abandon due to a trademark lawsuit from the Palm company), Windows PocketPC (PPC), Windows Handheld PC (HPC), Windows Mobile, WindowsPhone, and then Windows 10 Mobile. The HPC version was for the horizontal devices with a full keyboard, like shown in Figure 3.3 above. The PPC version used a vertical screen with a variety of on-screen virtual keyboards for text entry (see Figure 3.21). As with the Palm, all of these devices were designed



(a) (b)

Figure 3.21 (a) Hewlett Packard Jornada 540 Color Pocket PC from 1999. (b) HP iPAQ 4150 Pocket PC. Sources: (a) https://www.amazon.com/Hewlett-Packard-Jornada-Color-Pocket/dp/B00004TCF3. (b) From Amazon: https://www.amazon.com/HP-iPAQ-4150-Pocket-PC/dp/B0000TFCFO.

to be used primarily with a stylus. The Compaq iPAQ (Figure 3.21(b)) became very popular due to its relatively high-powered processor and ability to accept add-on cards. For example, my research group bought a sleeve so we could use the early WaveLAN cards (Figure 3.17) with the iPAQ to create the first mobile devices with Wi-Fi.

As mentioned above, a key goal of Windows CE was to be familiar to desktop Windows users while still being easier to use. Microsoft performed many user studies to help define and refine the UI [Zuberec 2000]. For example, they identified that the minimum size target for items to be tapped with a stylus was 5.04mm^2 and for a finger 9.04mm^2. Since different actions happen depending on whether the user taps or drags an object, they determined that 2mm was the distance required for the user to move before it was classified as a drag due to the inaccuracies of

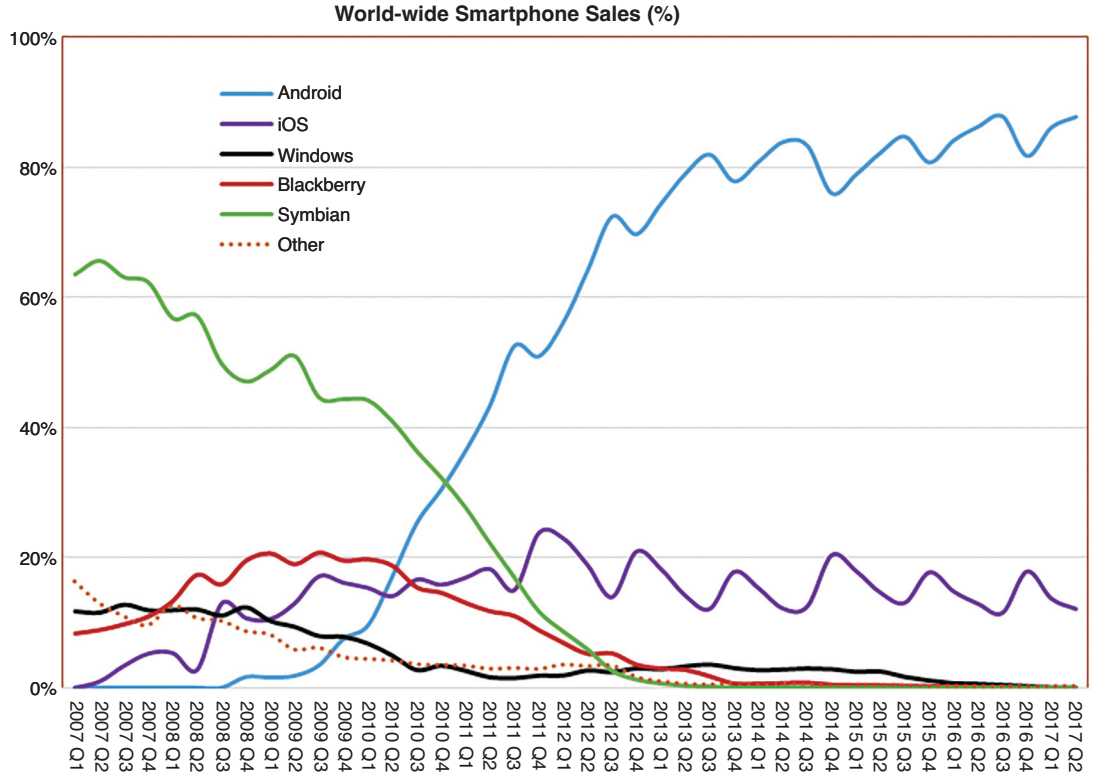


Figure 3.22 Worldwide smartphone sales (%), 2007–2017. Source: By Smartmo (Own work) [GFDL (http://www.gnu.org/copyleft/fdl.html) or CC BY-SA 3.0 (https://creativecommons.org/licenses/by-sa/3.0)], via Wikimedia Commons, https://commons.wikimedia.org/wiki/File%3AWorld_Wide_Smartphone_Sales_Share.png.

the sensor and the users' ability to tap accurately. Interestingly, they reported that although user studies identified the font Tahoma 10 bold as the best system font to use, they decided that it could not be used because not enough content would fit in the dialogs on the screen, so they selected Tahoma 9 instead [Zuberec 2000]. With respect to text entry, novice users did better with a regular on-screen keyboard, but experts preferred to use their character hand printing recognizer, which was like Graffiti but called *Jot*. A tradeoff they identified was that the initial designs achieved "walk up and use," but experts found it to be too slow because too many taps were required for common actions. Similarly, doing a double-tap, which is often used on desktop Windows, was found to be difficult and unnatural to perform with a stylus. This is another example where "consistency worked against learning and use" [Zuberec 2000].

In the end, Windows CE became a reasonable competitor in the PDA space, but Microsoft was never able to make an impact in the smartphone market—so as soon as PDAs faded in favor of smartphones, the various flavors of Windows for mobile devices faded out as well (see Figure 3.22).

3.15 RIM Blackberry

While the previous systems were broadly targeted at consumers in general, the Blackberry products from Research in Motion (RIM) were primarily targeted at business and government. The initial goal was to optimize reading and sending of pager messages and email. The first device came out in 1999 and featured two interaction techniques that would define Blackberry devices for many years: a multi-function rocker switch (also called a roller dial) and a tiny keyboard designed to be operated with one finger or two thumbs. The device did *not* have a touchscreen, so moving the rocker back and forth was the main way to navigate, and pressing in the rocker switch was the selection action (see Figure 3.23). The Blackberry devices originally used a proprietary network that was encrypted so companies could be sure their conversations would not be eavesdropped. Later, as data speeds increased, they switched to being smartphones and used the regular data networks. As shown above by the Palm Treo (Figure 3.20(b)), many other devices adopted the Blackberry-style thumb keyboard. After the iPhone became popular, RIM tried to release a full-screen touchscreen device with no physical keyboard, but it was awkward to use because it tried to be backward compatible with previous non-touchscreen devices. Another problem was that RIM was late to provide good-quality APIs with which third parties could build apps for the devices. Consequently, RIM's market share dropped to almost zero (see Figure 3.22), and the device was discontinued altogether in 2022 [Brooks 2022].

(a) (b)

Figure 3.23 (a) RIM Blackberry 850 from 1999. (b) Blackberry 7210 (first color model) from 2002. Both have the roller dial highlighted. Source: By Smartmo (Own work, GFDL or CC BY-SA3.0), via Wikimedia Commons, https://commons.wikimedia.org/wiki/File%3AWorld_Wide_Smartphone_Sales_Share.png.

3.16 Windows TabletPC

Microsoft continued its quest to have a credible product in the pen computing space with the TabletPC (Figure 3.24), announced in 2001 with a published specification, based on Windows XP, with the first devices appearing in 2002. Bill Gates famously predicted in 2002 that the TabletPC would be big—but it never was. Like previous pen computing devices, the expectation was that people wanted to handwrite to enter text because it was more "natural." So, the TabletPC supported handwriting recognition, which was much improved over what was available for the Apple Newton but still not sufficiently accurate. Also, it turns out that people don't really *want* to handwrite anymore. It is slower than typing for most people, and many people cannot even read their own handwriting, so one wouldn't expect a computer to be able to do it accurately. Given that it was going to be inaccurate, one might expect that there would be great interaction techniques for allowing users to check and correct recognition. But the commercial systems actually made correcting surprisingly difficult, even though Microsoft Research (and many others) explored interesting ways to foster corrections (e.g., Shilman et al. [2006]). Another big problem for TabletPCs is that besides text entry the rest of Windows had the same UI as the desktop version, but the small icons and widgets are very difficult to tap accurately with a pen, which has lower accuracy than a mouse (see Chapter 4).

(a) (b) (c)

Figure 3.24 (a) TabletPC. (b) Bill Gates announcing that the TabletPC will be big. (c) Handwriting recognition on TabletPC. Sources: (a) Janto Dreijer, Public domain, via Wikimedia Commons, https://upload.wikimedia.org/wikipedia/commons/8/82/HP_Tablet_PC_running_Windows_XP_%28Tablet_PC_edition%29_%282006%29.jpg. (b) Image courtesy of REUTERS/Jeff Christensen.

3.17 iPod

Having been spooked by the failure and ridicule of the Newton (Section 3.8), Apple achieved much better design and usability for the iPod music player, first released in 2001. There were many portable music playing devices available on the market at the time, but the iPod dominated. One reason was that it used an actual tiny spinning 5GB hard drive that could hold 1,000 songs. But a key contribution was the innovative wheel to let users navigate through all of those songs (Figure 3.25). Rotating the wheel slowly allowed for accurate one-by-one navigation, but rotating the wheel quickly would zoom through large numbers of items. Apple carefully adjusted how this interaction technique worked so that users could intuitively and quickly navigate through even thousands of menu items. Another key innovation falls under what is now called *service design* [Polaine et al. 2013]—in 2003, Apple launched the iTunes store that made it much easier to purchase and move songs to the iPod. It was very inexpensive per song, usually cheaper than buying a physical CD, and had virtually all the songs people wanted to buy, which all helped move online music from something shady, mostly associated with illegally sharing music, to a legitimate way to listen. Thus, Apple designed the entire service of finding, previewing, buying, and moving music to the iPod.

Figure 3.25 Original iPod from 2001.

3.18 iPhone

The Apple iPhone was first released in 2007 and was a radical departure from the conventional wisdom of the time. As mentioned above (Section 3.15), the industry was moving to Blackberry-style physical keyboards, and the iPhone exclusively used a virtual keyboard—with no handwriting or hand printing recognition. All previous devices primarily relied on a stylus for high-precision tapping, but the iPhone screen used a different kind of sensing (called *capacitive*) that does not even notice when touched with a pointy stylus; instead, it can detect *multiple* finger touches that previous systems had not used. Since it was finger operated, all the targets had to be large enough for fingers, and high-precision pointing was difficult. The iPhone only had one main physical button for navigating home (see Figure 3.26) whereas all previous systems had used multiple physical buttons. These design choices made the iPhone much easier to learn and use for novices while still being relatively efficient for experts.

The iPhone took advantage of many well-established interaction techniques like pinch-to-zoom (Sections 2.5.1.11 and 5.9.2), flick-to-scroll (Section 5.9.1), and the use of icons on the desktop to launch apps. As with the iPod, they refined these interactions so that they were smooth and natural feeling. Apple also innovated

Figure 3.26 Original iPhone from 2007, with the physical "home" button at the bottom.

some new interactions for the iPhone like bouncing at the end of a scroll to clearly show that the user has reached the end of the list [Ording 2008] and swiping to wake the phone up [Chaudhri et al. 2010].

3.19 Android

The Android operating system and UI for mobile phones was unveiled in 2007, with the first phones appearing in 2008. Google bought the company that was developing Android and offers it free to phone manufacturers, which accounts for much of its popularity with them. Android is based on Linux and Java, and the code is mostly open source. There are now hundreds of thousands of different device types that run Android, with thousands of different screen sizes and a variety of physical buttons, making it difficult for app designers to make sure that their UI works well on all devices. Android phones originally had three or more physical buttons at the bottom, for at least the back, home, and menu functions, but these later became soft buttons on the screen, often represented by a left-pointing triangle, a circle, and a square (Figure 3.27).

Figure 3.27 Android phone screen showing three icons at the bottom for back, home, and menu. Source: Screen capture from May 2018.

Figure 3.22 shows how the market share of Android has skyrocketed as a percentage of all smartphones. Also interesting in the graph is that the iPhone (labeled iOS) has remained relatively stable at around 20%, even as it has dominated the design of interactions. You can also see the Blackberry and Windows phone lines go almost to zero. The green Symbian line represents Nokia phones, which also pretty much disappeared. So currently, it is all Android and iPhone.

3.20 iPad

Finally, it is interesting to consider the iPad (see Figure 3.28), released in 2010, especially compared to the TabletPC and other previous pen computing tablets. The obvious difference is that the original iPad did not even support a pen—like the iPhone, it was designed to be used with fingers. The interaction techniques were therefore like those on a phone rather than a regular desktop, unlike what the TabletPC tried to do. The iPad was primarily used as an *output* device, for playing media, browsing the web, and reading, and de-emphasized user input. Although users could type using the on-screen keyboard or external third-party keyboards,

Figure 3.28 First version of the iPad from 2010.

that was not the primary use-case. The iPad also benefited from running all the apps that had already been designed for the iPhone, and Apple made it relatively easy for developers to modify their iPhone apps to run on the bigger screens. This contributed to making the iPad a popular and influential device.

3.21 Chapter Conclusion

This chapter has summarized some of the more important and influential mobile devices over the years. Figure 3.29 shows a timeline of some of these systems, along with the systems from the previous chapter.

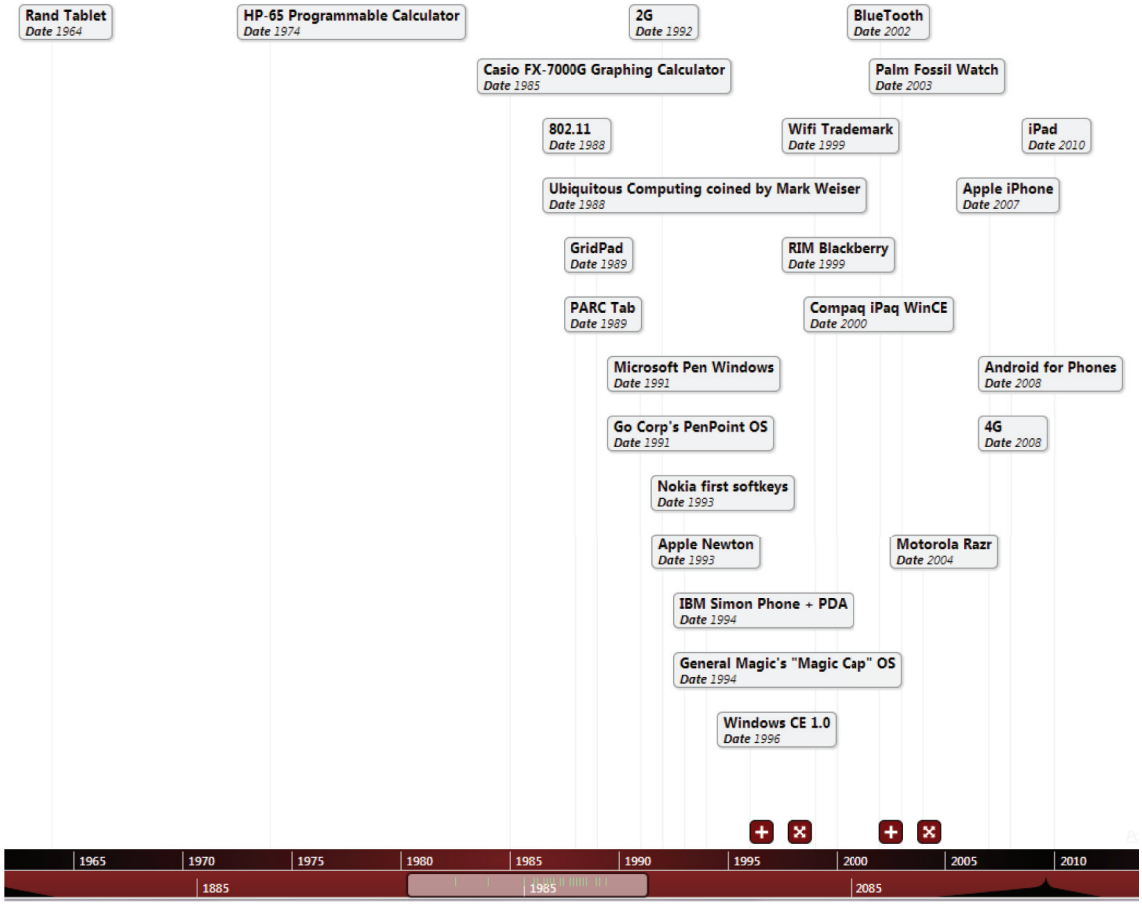


Figure 3.29 Timeline of some of the systems discussed in Chapters 2 and 3. Source: Chart I created using http://whenintime.com/tl/bradamyers/Handhelds/.

Obviously, many devices have been omitted. For example, I did not include any personal organizers, like the Casio or Sharp devices that were predecessors of PDAs, because they didn't have interesting interaction techniques or much influence on how the PDAs or smartphones behaved. I also didn't include any book readers, like the Amazon Kindle, or custom devices for vertical markets, like specialized warehouse or medical devices.

Using this history as context, Part II of this book discusses each of the interaction techniques in turn.

PART II

SPECIFIC INTERACTION TECHNIQUES

Part II of this book looks in detail at the various kinds of interaction techniques, with each chapter focusing on a different one. The chapters explain the history and design variations and the considerations and constraints that drive the designs. It is often difficult to convey the dynamics of how these interaction techniques behave, so the reader is encouraged to also view the "All the Widgets" video I produced that shows many of the older techniques [Myers 1990b].

CHAPTER 4

Pointing Devices

4.1 Introduction

A *pointing device* allows a user to indicate a particular location or object of interest, typically as a prelude to doing some operation. This is one of the most fundamental ways that a user must interact with any technology. This chapter discusses a wide variety of ways that users can do this, ranging from touch-sensitive screens to joysticks to a computer mouse to pointing a finger in the air.

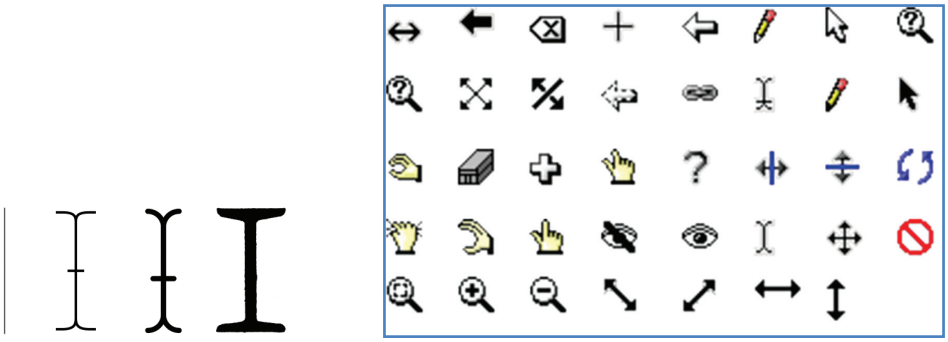
Note that this chapter only covers two-dimensional (2D) pointing devices like a conventional touchscreen or mouse. Three-dimensional (3D) pointing devices that can identify a point in 3D space are covered in Section 12.4, and one-dimensional (1D) input devices, like knobs or sliders, are covered in Chapter 5. The *ergonomics* of interaction techniques (how comfortable they are to use), including pointing devices, are covered in Sections 4.14 and 21.4, and special adaptations for people with disabilities are covered in Chapter 17.

There are many considerations when designing or choosing which pointing device to use, so this chapter also discusses the many dimensions on which devices can vary. For example, how does the device let the users know where they are pointing? How many states are supported—for example, does the device support hovering over the target without selecting? Finally, the key measures of *effectiveness* for any pointing device are the speed and accuracy with which users can point, and the chapter concludes with some standard ways to measure these.

4.2 Cursor and Selection

A *cursor* is displayed to provide feedback about the location on the screen. The word "cursor" is used both for the pointer position and for the text input position (Figure 4.1). On the early character terminals (Figure 2.7), the text position was the *same* as where the user was pointing, but in most modern systems the place where the next character will be typed, often shown by a blinking vertical bar, is *different* from the pointer location, often shown with an arrow (see Figure 4.2)—but the same word is used for both. I will use *text cursor* and *pointer cursor* to disambiguate

Figure 4.1 Two *cursors* at the same time: the text cursor is between the “r” and “s” letters, and the pointer cursor is pointing at the “o.” Source: Screen capture in May 2018.



(a) (b)

Figure 4.2 Various images (a) for a text cursor and (b) for the pointer cursor. Sources: (a) screenshot. (b) From: https://wiki.inspired-lua.org/File:Cursors.png.

when necessary. Sometimes people talk about the *mouse cursor* to mean the same thing as the pointer cursor, but the pointer cursor can be used with a variety of input devices in addition to a mouse device, so *mouse cursor* seems to unnecessarily limit what kind of pointing device can be used. The picture that is shown for the text cursor is usually a vertical bar, sometimes with horizontal or diagonal lines at the top and bottom (Figure 4.2(a)), whereas the pictures for the pointer cursor can vary widely, usually to provide feedback to the user about the current mode (Figure 4.2(b)). For touchscreens, like on the iPhone, there is usually no pointer cursor shown on the screen at all, but there will still be a text cursor.

The notion of the item that is *selected* is different from the pointer or text cursor. Typically, the *selection* is one or more items that will be affected by the next command, whereas the cursor identifies a location (e.g., the text cursor is usually shown *between* two characters, but older systems equated the text selection with the text cursor). On systems where the user can only move from item to item, like on most consumer electronics, the highlight of the current item is often called the *cursor*, even though that item is selected (e.g., Sprint Nextel [2006]). Interaction techniques for manipulating the textual cursor and text selection are covered in Chapter 8, and selection of graphical items is covered in Chapter 10. Moving the pointer location is covered in the rest of this chapter.

4.3 Mouse Buttons, Press, Hold, Click, Tap, and Drag—Definitions

Most pointing devices have one or more *buttons* with which the user can indicate that the current cursor position is of interest. When the user pushes down on the button, that is called *pressing* the button *down*, as opposed to *releasing* the button *up*, when it is let go. If the user presses and releases quickly, this is called a *click* of the button. The length of time to count as a click is usually user-settable, but it defaults to 500ms (half a second) in Microsoft Windows. If the user holds the button down for longer than that, it is called a *press-and-hold*. Apple introduced what it called *3D-touch* in 2015 on the Mac touchpads and iOS devices, which was basically just pressing hard, but it proved difficult for users to perform differently than press-and-hold, and by the iPhone 11 in 2019, 3D-touch was being phased out [Musil 2019].

If the user presses down the button and *moves* before releasing, this is called a *drag* since it is often used to drag an object on the screen, until the mouse is released. If the user clicks and then clicks again within a short period of time, this is called a *double-click* (and again the length of time between the two clicks is user-settable with a default of 500ms). Note that it is possible to double-click-and-hold, by doing down–up–down within 500ms, but then holding the mouse button down after the second press. Similarly, the user can do double-click-and-drag. *Triple*, *quadruple clicks*, and so on are also possible. When there are multiple buttons, all of the above descriptions may be preceded by which button was used, such as *left-down* versus *right-down*. Often, the meaning of the mouse button will also depend on whether some keyboard keys are held down, so all of the above may be preceded by a keyboard key modified, such as Shift, Control (Ctrl), Alt, Command, and so on, like "Alt-left-down" or "Shift-click." As discussed below, there are a variety of devices with which a user may point, and those with an on-screen cursor and a button, including mice, lightpens, and touchpads, use this terminology.

For touchscreens, however, the user bringing their finger into contact with the screen is often called a *touch*, which may also have the same modifiers as above, such as *touch-down*, *touch-up*, *touch-drag*, *touch-and-hold*, and *touch-and-drag*. (Actually, JavaScript calls it `touchStart` and `touchEnd`, which is inconsistent with its use of `mouseDown` and `mouseUp`!) A quick touch-and-release is called a *tap* (rather than a *click*), and *double-tap* and so on are also used. If the touchscreen can support detecting multiple fingers, then these may be modified by the *count* of how many fingers so far, such as the *first-touch* or *second-touch*, since no popular devices can determine *which* finger was used (so they cannot differentiate touching with the index finger vs. the thumb).

Note that the touch and press terminology is often used interchangeably (even in this book!), especially for applications that can run on systems that support both

a mouse and touchscreens. For programmers, the operating system may actually generate both mouse *and* touch events, so either kind of event can be used in the code (see Section 20.12.2).

4.4 Lightpen

The *lightpen* (sometimes spelled "light pen") was an early pointing device, starting in the 1950s, as mentioned in the discussion of Sketchpad (Section 2.4.1) from 1963. The lightpen had a little camera in the front that is watching for light coming from the screen (Figure 4.3). Remember that Sketchpad used a vector scan screen (Section 2.3) where the beam continuously draws the picture. The computer used the camera in the lightpen to watch for *when* it saw the beam, and since it knew when it told the beam to be at each place, it could tell where the pen was pointing. The cursor shown at that point was typically a bright spot (shown in Figure 4.3) to make it easier for the lightpen camera to see it.

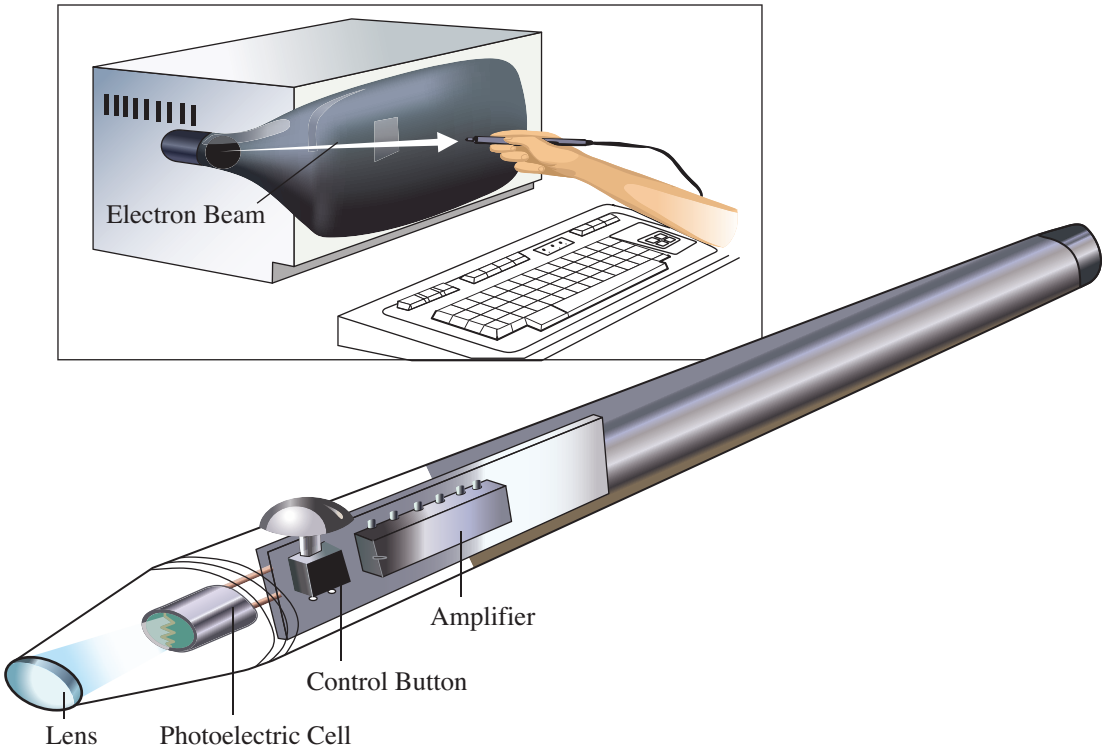


Figure 4.3 The lightpen. Source: Copyright © 2017 by Wayne E. Carlson, https://ohiostate.pressbooks.pub/graphicshistory/chapter/3-5-input-devices/.

There were many disadvantages to the lightpen. It was relatively low resolution, and it only worked when pointing at a part of the drawing—not at a part of the screen that was dark. Therefore, interactions usually started with the pen pointing at something already displayed, and moving slowly away, since moving quickly often caused the camera to lose sight of the drawings. A key problem, which is shared by many devices requiring pointing in the air, and touchscreens mounted vertically, is arm fatigue—it is tiring to hold one's hand in the air for long periods of time. This has been called *gorilla arm syndrome*, allegedly because the user looks like a gorilla while pointing at the screen.

4.5 Arrow Keys

Computer keyboards have almost always included *arrow keys*, also called *step keys* (a notable exception is the original Macintosh, which famously eschewed arrow keys to force users to use the mouse, see Figure 2.16). Originally laid out like a plus sign, the current arrangement for keyboards (Figure 4.4(a)) was found to be more usable. However, the plus sign arrangement is still used with remote controls for consumer electronics and on game controllers (Figure 4.4(b) and (c)). Step keys are also frequently used in automobiles to control the navigation/entertainment system, or a knob might move the selection forward and backward (Figure 4.5).

(a) (b) (c)

Figure 4.4 Arrow keys on (a) a keyboard, (b) a remote control, and (c) Nintendo Switch pro-controller. Source: (a) By Lars H. Rohwedder (User: RokerHRO)—Selfmade picture, Public Domain, https://commons.wikimedia.org/w/index.php?curid=2227002. (b) By Altaïr Skywalker 47—Took from camera, CC0, https://commons.wikimedia.org/w/index.php?curid=62415280. (c) By Evan-Amos—Own work, Public Domain, https://commons.wikimedia.org/w/index.php?curid=61803234.

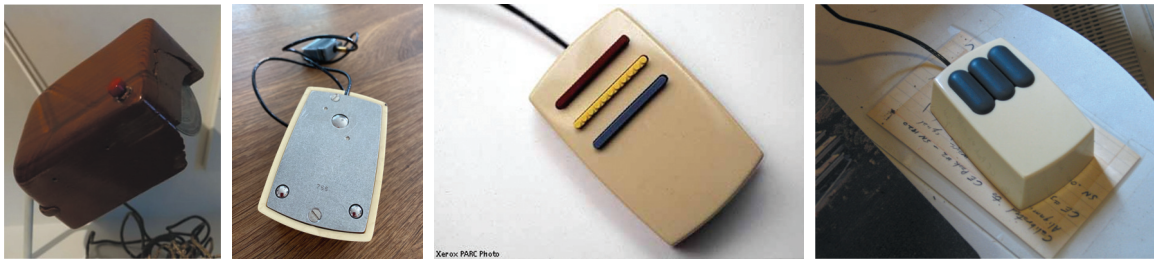
Figure 4.5 Mazda automobile control knob can be rotated to move the selection like step keys or pushed in different directions as a joystick. The semi-circle button on the bottom left is for "back" or "abort." Source: Photo by author.

Although mostly thought of as a way to move the text cursor (Section 8.6) or to change the selected item or character direction, arrow keys can also be used to move the pointer. For example, in Microsoft Windows (pretty much all versions), if you press Alt-Space, followed by the "m" key (for "move"), you can then use the arrow keys to move both the current window and the pointer cursor.

4.6 Mouse

As discussed in Section 2.4.2, the computer *mouse* was invented at SRI by Bill English and Doug Engelbart around 1967 [English et al. 1967] and used in the famous demonstration of NLS the following year [Engelbart and English 1968]. The first version was made out of wood (Figure 2.9(a)) and used two roller wheels mounted perpendicularly (Figure 4.6(a)) and one to three buttons on top (Figure 2.9(a)). When Xerox PARC developed their version of the mouse, they used a roller ball on the bottom (Figure 4.6(b)), which was more reliable. The original mouse at PARC had three buttons arranged horizontally and labeled by color (Figure 4.6(c)), but it changed to be three vertical buttons (Figure 4.6(d)), which were more comfortable to reach and easier to differentiate by feel without looking.

Xerox did a number of studies comparing the mouse to other available pointing devices of the time and found significant time and accuracy advantages for the mouse [Card et al. 1978]. Besides being fast and accurate to use, the mouse has many other advantages. Users can rest their arms and hands on the desk surface, which is much more relaxing than holding your arm in the air for a lightpen or



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

Figure 4.6 (a) Rollers on the original SRI mouse. (b) The Xerox mouse used a roller ball and (c) originally had three vertical buttons: red, yellow, and blue. (d) A later Xerox PARC design with three buttons side-by-side. Source: (a) Image courtesy of The National Museum of American History. (b) Vincent Lextrait, CC-BY-SA 4.0, https://commons.wikimedia.org/wiki/File:Xerox_Alto_mouse_bottom.jpg. (c) Xerox PARC. (d) Marcin Wichary, CC-BY-SA 4.0, https://commons.wikimedia.org/wiki/File:Xerox_Alto_mouse.jpg.

touchscreen. Another important property is that the mouse can be engineered so the pointer does not move when a button is pressed, since force is aimed downward, which means that the cursor stays steady when users press and release the buttons, increasing accuracy. In contrast, the button on the side of a lightpen (Figure 4.3) tends to push the pen to the side.

When the mouse is moved across the surface, typically of a desk, the pointer cursor moves on the screen. If the mouse gets to the edge of the comfortable moving area, the user just lifts the mouse off the desk, moves it back, and puts it down, which is called *clutching*, by analogy to a car clutch which connects and disconnects the motor from the wheels. While the mouse is in the air, the cursor does not move on the screen. The distance that the cursor moves on the screen does not map one-to-one with how far the mouse is moved on the desk. The mapping is called the *gain function*, which can be adjusted on all modern systems (see Figure 4.7). When the user moves the mouse slowly, the cursor moves slowly as well, but when the user moves quickly, the mouse moves much further per unit distance of the mouse's movement on the desk. Bill Atkinson said that this was added to the Lisa when they decided to move the menu bar to the top of the screen, to make moving there faster (Section 28.2).

4.7 Joysticks and Trackballs

Joysticks had been used for years in aviation, and it was natural to investigate their use with computers. Mostly, they have been applied to control games (Figure 4.8(a) and (b)) but are also used in automobiles (Figure 4.5) and some remote controls

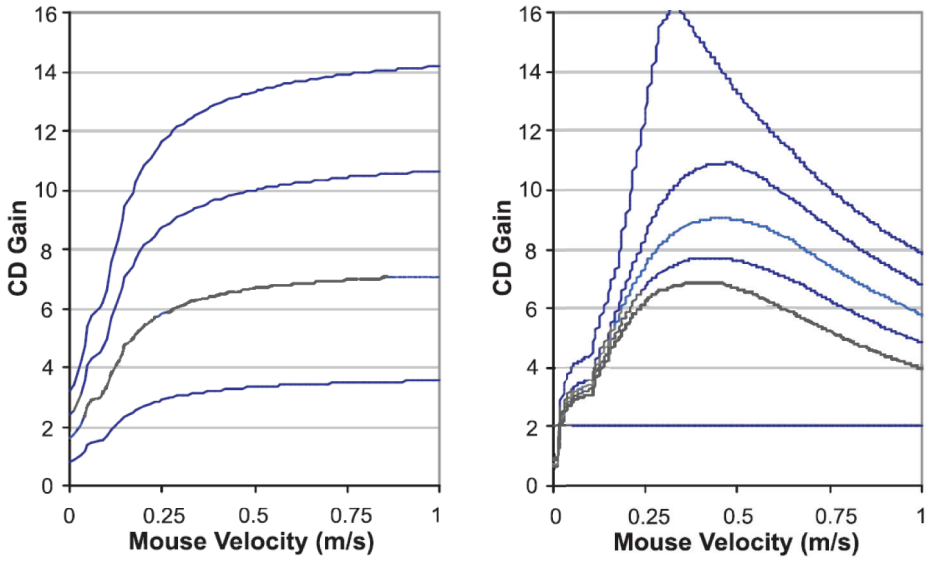


Figure 4.7 Mouse gain functions for Windows and Mac, as calculated from measurements and the source code [Casiez et al. 2008]. Source: Fig. 3 of Casiez et al. [2008].

Figure 4.8 (a) Old game joystick. (b) Joysticks on a game controller and (c) on an Epson remote control. (d) Position-controlled joystick. Sources: (a) Image from iStockPhoto. (b) and (c) Evan-Amos, Public Domain, https://commons.wikimedia.org/w/index.php?curid=61803234. (c) Photo by author. (d) Junkyardsparkle, CC0 1.0, https://commons.wikimedia.org/w/index.php?curid=24353055.

(Figure 4.8(c)). There are two kinds—most joysticks are *rate-controlled*, which means that the further the user pushes the stick away from the center, the faster the position moves in that direction. Since the joystick is essentially measuring the force at which it is being pressed, these are sometimes called *force-sensitive*. The stick has a spring that brings it back to center when the user releases, and being in the center means the position does not move. An extreme form of this is called an *isometric joystick*, when the stick doesn't actually move at all—it just measures the force in different directions. A variation is where the joystick only has one position away from the center, so it is either pressed in that direction or not, which is commonly used for automobile joysticks (Figure 4.5).

An alternative, which is rarely used, is *position controlled*, where the stick does not have a spring and can move anywhere within a box or circle, and the position is determined by the position of the stick (Figure 4.8(d)).

A *trackball* is a similar device where the user rolls a ball around to move a cursor on the screen (Figure 4.9(a)). All trackballs are relative devices like a mouse, where the cursor moves a certain amount based on the distance the ball is moved, with the amount varying by a gain function based on the speed of movement. Trackballs are typically used in some games (see Section 12.5.1), but they were also used in the original Macintosh Portable (1989) and PowerBooks (1991) and a few other laptops,

(a) (b)

Figure 4.9 (a) Kensington trackball. (b) Blackberry Pearl has a tiny trackball below the screen. Source: (a) By Suimasentyottohensyuushimasuyo—Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=1509949. (b) By NeonGenesis-Pop (talk)—Self-made, Public Domain, https://en.wikipedia.org/w/index.php?curid=15765684.

before the trackpad became ubiquitous, and in the Blackberry Pearl phone (from 2006 to 2010, see Figure 4.9(b)). Trackballs are also popular as a mouse replacement for people with certain physical disabilities since they tend to be more stable and use different fingers and movements compared to a mouse. (For example, I wrote this book using the Kensington Orbit trackball instead of a mouse due to tendonitis in my right index finger, see Section 17.3.1.)

4.8 IBM TrackPoint

The *IBM TrackPoint* (also called a *pointing stick*) is a rate-controlled isometric joystick that is placed between the G, H, and B keys (see Figure 4.10) in the center of IBM and Lenovo laptops (among others). Chapter 34 examines how it was developed by its inventor, Ted Selker. In summary, IBM did eight years of human factors work to refine the device, including many experiments about the material of the red cap, the placement of the buttons below the space bar, and the gain function mapping how hard one presses to the speed of the cursor movement.

4.9 Graphics Tablets

A *graphic tablet* (also known as a *digitizer*, *drawing tablet*, *digital drawing tablet*, *pen tablet*, or *digital art board*) is an input device that measures the position on the surface of one or more styluses. I am differentiating these from touchscreens (Section 4.10) in that graphics tablets are not screens and have no output. They also generally require the use of a special stylus or pen rather than a finger. They are typically placed on a desk and are relatively large so that users will often rest their hands or arms on the tablet, which is why they are not touch sensitive and

Figure 4.10 IBM TrackPoint joystick and its buttons. Source: By Takeshi Kuboki from Amagasaki, Japan—ThinkPad USB トラックポイントキーボード Uploaded by clusternote, CC BY 2.0, https://commons.wikimedia.org/w/index.php?curid=26710722 (excerpt).

Figure 4.11 Rand tablet in use. Source: Image courtesy of the Rand Corporation, http://www.rand.org/content/dam/rand/pubs/research_memoranda/2005/RM4122.pdf.

instead often use some kind of magnetic or electronic means to detect the location of the pen. As mentioned above, one of the first graphics tablets was created by RAND Corporation in 1964 (Figure 4.11) and was used to develop early gestural and hand printing recognition algorithms (see Sections 2.5.1.11 and 14.3). Later versions by companies such as Summagraphics and Wacom have been popular for CAD/CAM and artists to make precise drawings and to give commands to the system (Figure 4.12).

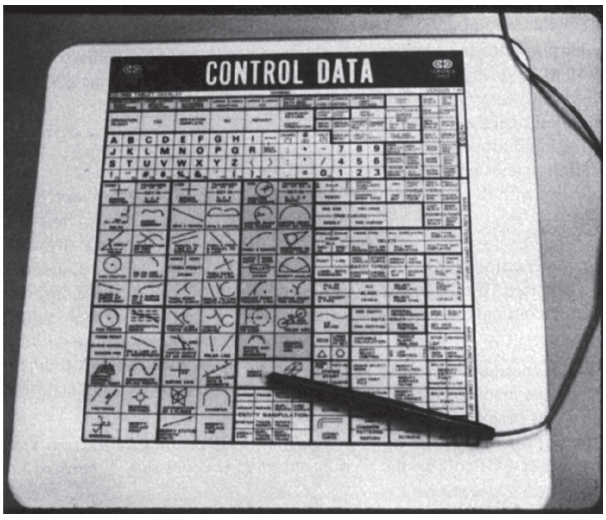


Figure 4.12 Digitizer tablet with printed overlay showing specific locations for commands. This must be used in *absolute* tracking mode. Source: Fig. 11 of Foley et al. [1984].

In addition to measuring the pen's position, tablets can sometimes also measure the *force* at which the pen is pressed down (sometimes with a click to indicate two specific levels of pressure, with pressing harder representing the same thing as the mouse button pressed) and the *angle* and *rotation* of the pen with respect to the tablet, all of which have been used to control various interactions. The pens often have one or more buttons on the side (now usually two) that can be pressed with the index finger to provide signals like mouse buttons. Some tablets, especially those using magnetic sensors, can detect when the pen is *hovering* above the tablet, as opposed to touching the tablet, providing another level of state (hover, touching, and whether the button is pressed).

4.10 Touchscreens

Touchscreens are now ubiquitously used on mobile devices and even laptops and large screens. As discussed in Section 3.2.2, touchscreens originally were intended to be used with a pointy stylus and later with one or more fingers. Note that the size of the touchscreen may have a big impact on how it can be held and used. Whereas a phone-size device can be held in one hand from the sides, for a tablet-size device the users' fingers must wrap around to the front (Figure 4.13(a)). Thus, tablets must have an inactive area on the front for holding them, unlike phones where the display can extend to the edge.

Some touchscreens can be quite large, from tablet size up to whiteboard size, and may be oriented vertically or horizontally for use by multiple people. These wall-size or table-size touchscreens (Figure 4.14) may be deployed in classrooms, museums, meeting rooms, and so forth. Larger touchscreens may also have the problem of users resting their hands on the surface (Figure 4.13(b)). Other touchscreens are quite small, such as on a smartwatch (Figure 3.19), or the long and skinny *Touch Bar* on some Apple MacBooks (Figure 4.15).

An important issue with touchscreens is the accuracy of the touch, which is affected by many things. First, the accuracy of the underlying sensor may be low. Even if it has relatively high accuracy, it may have *jitter* so that the measured location changes even when the pointer (finger or stylus) is held steady, which also lowers the useful accuracy (see Section 4.14). When touching with a finger, the pad of the finger is larger than many targets (like individual characters). Another issue is that the finger or even a stylus may block the view of the desired target (which is why the iPhone at one point used a popup bubble to show what was under the finger (see Figure 8.37(a))). A further issue is *parallax*, which is that the position may look like it is in a different position than it is due to the bending of light. This is a result of the glass (or plastic) of the screen since the display is on the bottom but the touch sensor is on the top. If a stylus or pen is held at an angle, the user

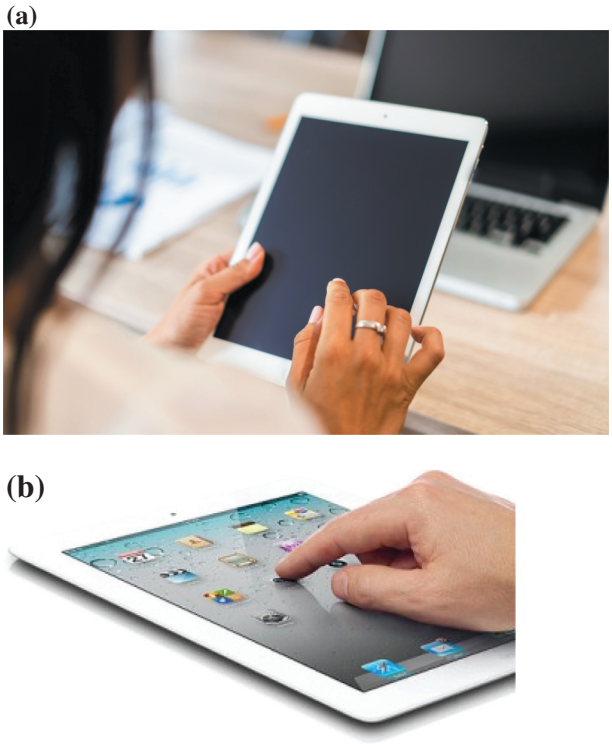


Figure 4.13 (a) Holding a tablet requires there be an inactive area for the fingers on the front (but notice that this user is still touching the screen with her left thumb). (b) Resting your hand on a tablet can confuse the touchscreen. Sources: Photo 126525196 © Opolja | Dreamstime.com.

(a) (b)

Figure 4.14 Wall and table-size touchscreens. Sources: (a) svonog, CC BY 2.0, https://commons.wikimedia.org/w/index.php?curid=2931288. (b) Alamy photos.

(a) (b)

Figure 4.15 (a) Small Touch Bar on MacBook is above the keyboard. (b) Four examples of what it may show. Source: Image courtesy of Apple Inc., https://support.apple.com/en-us/HT207055.

might think that the touch point continues on the vector through the pen to the back of the glass, whereas it is actually where it touches on the front.

4.11 Remote Interaction

There are a few technologies to allow people to interact with screens at a distance. *Gaze tracking* uses where the user is looking by detecting the direction of the user's head and location of the pupils, usually with an infrared light and camera. A key problem with using eye gaze as an input device for pointing is that the user looks all around to understand the screen, and it is very hard to accurately look at small targets, so the resolution is poor; it can still be useful for user studies

(a) (b)

Figure 4.16 Pointing across the room using (a) the Wii Controllers or (b) finger detection by the Kinect. Sources: (a) DepositPhotos. (b) The Conmunity - Pop Culture Geek from Los Angeles, CA, USA, CC BY 2.0, via Wikimedia Commons. https://commons.wikimedia.org/wiki/File:CES_2012_-_Microsoft_Kinect_Star_Wars_Episode_1_Podracing_(6764013293).jpg.

(to determine where the user is looking) or for people with disabilities (see also Section 17.4.1).

For interaction with large screens, sometimes it is not convenient to touch them, so researchers have investigated detection of the position of a laser pointer on the screen [Myers et al. 2002a]. More practical are devices like the Nintendo Wii controller and the Microsoft Kinect (Figure 4.16) that allow users to point to the screen from across the room. However, all of these techniques are quite inaccurate and tiring to use for an extended period (see Figure 4.19). There are also remote-control devices with joysticks (Figure 4.8(c)), which are slow and awkward to use.

4.12 Dimensions for Pointing Devices

The pointing devices that have just been discussed can be classified according to various dimensions, which helps clarify options available to designers of new devices or users selecting a device.

4.12.1 Direct versus Indirect

With a *direct pointing device*, the user points to the actual point of interest on the screen. Examples include the lightpen and touchscreens. In this case, there is often no need for a pointer cursor since the tip of the finger or pen shows where the user is pointing. This is more natural and has been tested to be the fastest pointing method [Myers et al. 2002a], but it does have disadvantages. As mentioned above (Section 3.18), a finger touch is relatively low resolution, but some devices have relatively low resolution for detecting where a pen is—for example, the pen on the TabletPC (Section 3.16) had a lot of jitter, making it difficult to select small targets like single letters or icons. The solution to jitter is usually a *low-pass filter* that takes the average of the last few values coming from the device—this removes high-speed jitter but it can also make the pointer location lag behind the finger or stylus especially when the user is moving quickly. Another problem with direct input is that the finger or stylus may block the view of what the user is trying to point to (see also Section 8.6.3.3).

With an *indirect pointing device*, the user's movement is somewhere other than on the screen. Examples include a mouse or touchpad or pointing in the air. These typically require a pointer cursor be shown on the screen to show the location, which has the advantage that the picture in the cursor can be used to provide feedback (see Figure 4.2). Indirect pointing usually requires some training and practice (although nowadays probably anyone who is tech-savvy has learned to use a mouse and/or touchpad).

The first design choice for an indirect pointing device is the direction that the cursor moves when the device or finger is moved, especially since the devices are usually perpendicular to the screen. Currently, moving the mouse away from the screen causes the cursor to go down, which seems natural if you think of the screen being laid flat face up. However, if you think of the screen as a mirror, then consider moving the mouse or fingers on a touchpad toward the screen—this would cause the image of the mouse to move *down* the screen. However, this option is never used (although Apple recently switched the direction of *scrolling* using the touchpad, as discussed in Section 5.9). The next design choice is the *gain function*, which maps the distance moved on the device to the distance moved by the cursor, as discussed above (Section 4.6).

4.12.2 Absolute versus Relative

A direct pointing device always uses *absolute coordinates*, which means that the mapping between the pointer position and the cursor position is one-to-one. A mouse device always uses *relative coordinates*, which means that the movement of the pointer controls the movement of the cursor, and the absolute position of the device is not relevant (picking a mouse off the table and putting it back down somewhere else does not move the cursor). However, indirect devices like touchpads and graphics tablets can be *either* absolute or relative. Most touchpads are used in a relative mode, so moving in a direction moves the cursor in the same direction, and where one puts down one's finger is not relevant. However, it is possible for a touchpad to be absolute by mapping the top-left of the touchpad or graphics tablet to the top-left of the screen, and the bottom-right to the bottom right of the screen, and then mapping wherever the touch happens to the corresponding point on the screen. If the touchpad or graphics tablet is used for *digitizing*, which means to copy a drawing from paper, then this absolute mode would be preferable. Figure 4.12 shows a graphics tablet in absolute mode used for selecting commands, which are printed on a piece of paper overlaid on the digitizer.

4.12.3 How Many States Are Supported?

When you move the pointer cursor from a mouse over most widgets without pushing any of the buttons, the widgets usually highlight to show that they are active (Figure 1.3(a)). This is called the *hover* state and is inherently supported by all indirect pointing devices (since the cursor's position indicates the currently hovered object, and the cursor must be able to move across the screen without activating everything). A few direct devices can be detected when they are hovering and not touching the screen. For example, some touchscreens (like some brands of Smartboards by Smarttech.com and an early multi-finger sensor [Rubine 1991a]) use

cameras placed in front of the screen to watch for the fingers to be in vicinity, which can then detect hovering. Similarly, some styluses use magnetic detection that can identify the pointing device *before* it touches the screen. However, most touchscreens in use today on smartphones and tablets do *not* support a hover state, so interaction techniques must be designed to *not* depend on hover.

The cursor for a mouse is always on the screen, so there is always a location for the mouse pointer. Early tablets, such as the TabletPC (Figure 3.24), also had a pointer cursor. However, modern touchscreens on smartphones and tablets do *not* report a location at all when the finger or stylus is not at the screen, and there is generally no pointer cursor shown. Therefore, these touchscreens may only have two states: one for when the finger or stylus is detected and one for when not.

For mouse, touchpad, and some other systems, a related question is how many buttons are supported? The original Xerox mouse had three buttons (Figure 1.9), the Star mouse had two buttons (Figure 2.14), and the Macintosh mouse famously had only one button (Figure 2.16). The Star developers had performed significant user studies on how many buttons to include and found that people seemed to best understand and be efficient with two [Bewley et al. 1983], whereas Apple decided that with only one button users would not be confused about which button to hit (see Section 23.9). Currently, one can buy a mouse with almost any number of buttons (Figure 4.17(a)), but the "regular" PC mouse has two main buttons (called *left* and *right* buttons) and a scroll wheel in the middle (see Chapter 5). The scroll wheel (Figure 4.17(b)) can actually serve as a button since pressing down on it sends a

(a) (b)

Figure 4.17 (a) Razer Naga 2014 mouse has lots of buttons. (b) Regular mouse with scroll wheel. Sources: (a) By osman gucel—Razer Naga 2014 MMO Gaming Mouse, CC BY 2.0, https://commons.wikimedia.org/w/index.php?curid=50914415. (b) By Darkone—Own work, CC BY-SA 2.5, https://commons.wikimedia.org/w/index.php?curid=235633.

middle mouse button down signal, which some programs may respond to (e.g., see Section 5.8). Some scroll wheels actually have two more button signals—you can push the scroll wheel to the left and right (without turning it or pressing it down), which on a PC may scroll left and right by one increment. Mice for Macintosh still have one button, but you can get the right-button action using a keyboard key while pressing the mouse button.

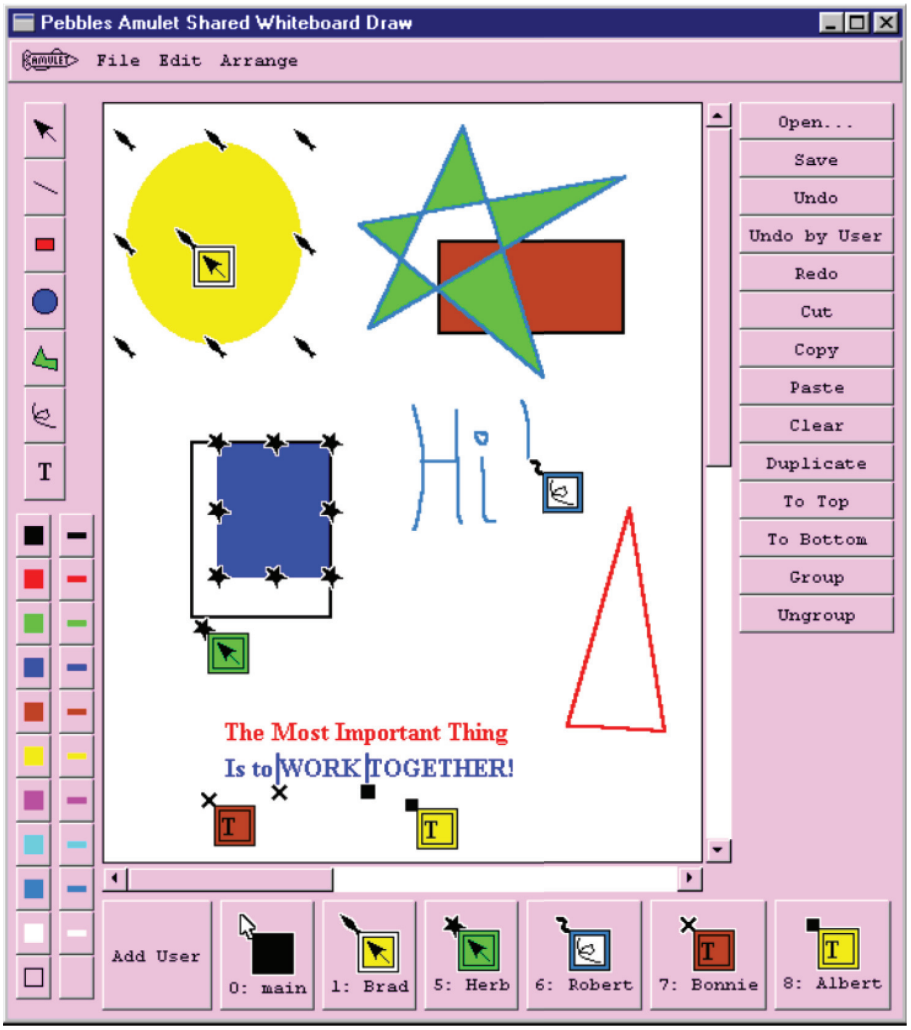


Figure 4.18 My PebblesDraw application, which can support up to eight users at a time, each with their own cursor shape. The user's cursor shape is also used to indicate that user's selected object. Source: Myers et al. [1998].

Some styluses and lightpens can have one or more buttons on the side that can be hit with a finger (Figure 4.3), but pushing this button will often tend to push the pen to the side. Another design for a stylus has active areas at both ends, and the user can typically use one end to draw and the other to erase like a real pencil. Alternatively, some styluses have a spring-loaded tip and can detect how hard the user is pushing them into the screen or graphics tablet. Similarly, some touchscreens themselves can measure force, with different levels of force performing different actions. For example, the iPhone *3D touch* used pressure to trigger alternative actions than a regular touch.

Another dimension is how many pointing devices (or fingers) can be used at the same time. Section 3.18 discussed the detection of one or multiple fingers on smartphones. Modern desktop computers will let users attach multiple pointing devices, but they all move the same cursor. (I routinely attach a trackball to my Lenovo laptop and can switch among using the TrackPoint (Figure 4.10), the touchpad, or the trackball to move the pointing cursor.) We also looked at allowing people to point with their dominant hand while using a slider in their non-dominant hand, and showed that this can be faster and that people will use both hands at the same time naturally without prompting—but this has not caught on [Buxton and Myers 1986, Myers et al. 2000b].

Researchers have looked into having multiple cursors controlled by multiple input devices, such as MMM [Bier et al. 1992] and my PebblesDraw application (Figure 4.18) [Myers et al. 1998], but this has not caught on for desktops. On the other hand, for large screens on a wall or tabletop (Figure 4.14), having multiple people draw at the same time has been researched for years [Weiser 1993, Dietz and Leigh 2001] and deployed in many current commercial systems, such as the Microsoft HP6-00001 Surface Hub that comes with two pens. A key question for multiple touch is whether the system can tell which is which. For example, most touchscreens cannot tell which finger is which or if the fingers belong to one or multiple people, whereas the Microsoft Surface Hub can tell the two pens apart.

4.13 Other Issues

On modern operating systems like Windows and the Macintosh, the cursor is usually actively following the mouse or trackpad, even when the rest of the system seems to be hung and non-responsive. This is because the pointer tracking is handled at a very low level of the operating system and the hardware. Thus, seeing the cursor move shows that the hardware hasn't crashed, but it does not actually show that the application that the user is using is (or will be) responding.

Another interesting issue is *clickahead*—if the user clicks somewhere but the application is not ready to listen to the click, is it saved for later or thrown away?

Keyboard typing is almost always queued and saved for later (see Section 8.8), whereas the response to the mouse is quite varied. In Microsoft Windows, it depends on the application and what state the application is in, so sometimes the click will be processed later and sometimes not. This makes it dangerous for a user to just click repeatedly in the same place since if the first click causes what is under the cursor to change (e.g., by removing the window), then the second click might operate a newly visible control that the user didn't know was going to be there. Most touch-screen systems (like smartphones) do not queue the clicks. See also Section 11.4 about whether the click which changes the window that is on top or has the input focus is *also* queued for applications, and see Section 20.12.2 for a discussion of the *event model*, which is how queuing is implemented.

4.14 Testing for Speed and Accuracy

Although there are many dimensions and issues to consider when choosing a pointing device or designing a new one, the key requirements are the *speed* and *accuracy* with which the device can be used. This is limited by the properties of the device, as discussed above, but also by properties of humans and the muscle groups being used. It would not be surprising that a device operated by one's foot would be slower and less accurate than a device that uses one's fingers since the fingers are much more dexterous. It may be more surprising that psychologists have developed a model (basically a mathematical formula) that has been shown to be surprisingly accurate at *predicting* performance across a wide range of different kinds of pointing devices.

Using a pointing device has a number of steps, depending on the device, what the user was doing previously (e.g., if the user was typing), and the user's task with the device. Typically, users must (1) move their hands from the keyboard or wherever they are to where the device is, and then (2) *acquire* the device, which (for a pen) may involve picking it up. Often this step requires looking at the device, taking eyes off the screen (or off the road if the pointing device is used in a car). Then (3) the user must use the device to move to the general area of the target, which may be a fast, ballistic motion, especially if the target is far away. Then (4) the user must accurately get the device to point at the target, which is obviously harder if the target is small. This step is also subject to *overshoot* if the user moves too fast and misses the target and has to backtrack. Finally, the user must (5) signal that this is the correct location, often by tapping the screen or pressing a mouse button. For some kinds of actions, such as dragging or long-press, the user will then (6) move (or hold still) while still signaling (e.g., while still holding down the mouse button or while still keeping the finger or stylus on the screen), followed by (7) signaling that the interaction is finished—for example, by releasing the mouse button

or moving the finger or stylus off the screen. All of these steps take time and are subject to errors. Figure 4.19 shows a trace of some of the errors when trying to use a laser pointer at a distant screen as a pointing device, which is quite slow and inaccurate for many of the steps.

4.14.1 Fitts' Law

The key model used for measuring the speed and accuracy of pointing is called *Fitts' law*, named after Paul M. Fitts Jr., who developed the model in 1954 while studying human movements during his time in the US Air Force [Fitts 1954]. The key observations underlying this model are that people move faster when there is a long distance to move, but they slow down when they need to hit a small target. Therefore, the time to move a pointing device to tap on a target (and the number of expected errors) depends on the distance to be traveled and also on the size of the target. The original apparatus that Fitts used is shown in Figure 4.20. The target was the striped area in the middle and the other areas on each side were used to record misses (errors). Fitts measured four sizes of targets (2in, 1in, 0.5in, and

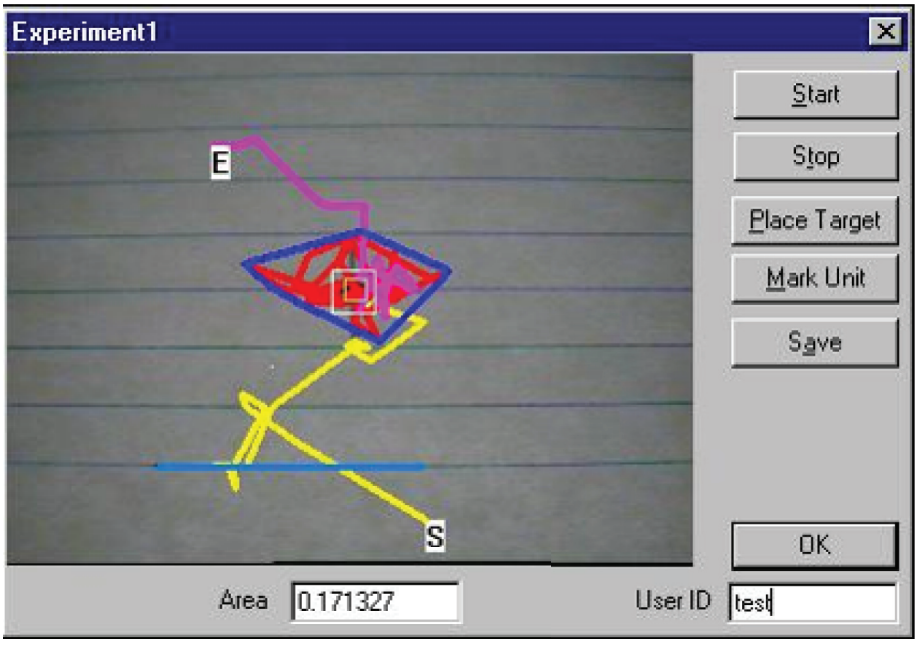


Figure 4.19 The user is trying to aim a laser pointer at the small square in the center. The beam is turned on at the spot labeled "S," and the yellow line shows the path while trying to get there. Then, the user tries to hold the point as still as possible for three seconds, resulting in the red trace. Finally, the user turns off the beam, which causes the beam to move along the pink path to "E" because the button is on the *side* of the laser pointer, so the pointer moves while the button is being released. Source: Myers et al. [2002a].

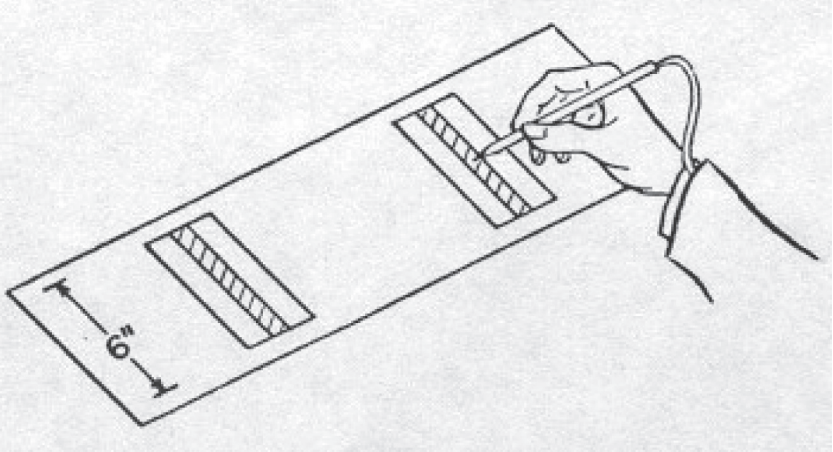


Figure 4.20 The original apparatus used by Fitts to measure throughput. Source: Fitts [1954].

0.25in) and four distances between targets (2in, 4in, 8in, and 16in) and confirmed the relationship.

As mentioned above (Section 4.6), Stu Card, Bill English, and Betty Burr did extensive studies in 1978 on various pointing devices including the mouse [Card et al. 1978]. They wanted to study expert behavior, but at the time there were no experts in using a mouse. Therefore, they had their participants use the devices for four to six hours across multiple days, during which they performed 1,200 to 1,800 clicks, in order to achieve mastery. The tasks involved using the pointing devices to reach a target in a text editor, and the time that it took was found to be *linear* with the distance when using step keys, whereas the time increased with the *log* of the distance for continuous devices like the mouse since users can move much more swiftly when there are large distances to move, and only need to go slowly as they reach the target [Card et al. 1978]. This is shown in Figure 4.21(a). When studying the effect of the size of the target, they found that the positioning time for both the mouse and the joystick decreased with the log of the target size (Figure 4.21(b)).

The overall formula is:

$$\mathrm{T} = a + b \log_2 (D/W + 0.5) \tag{4.1}$$

Where:

- T = time to complete a movement;
- a = fixed cost to start/stop moving and click;
- b = inherent speed of device; and
- $\log_2(D/W + 0.5)$ = "index of difficulty."

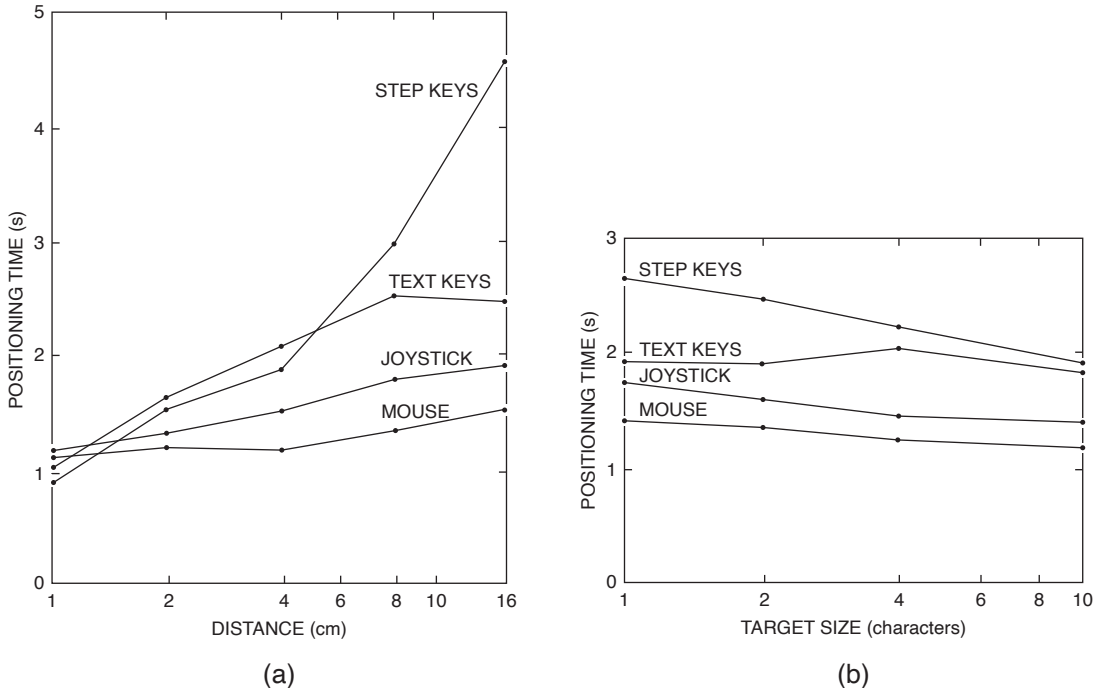


Figure 4.21 (a) The time to position a cursor varies linearly with the distance for step keys but by the log of the distance for continuous devices. (b) The positioning time for both the mouse and the joystick decreases with the log of the target size. Source: Figs. 3 and 4 of Card et al. [1978].

The *index of difficulty* gets higher with distance (D) and lower with width (W). Note that you get the same outcome if you double the distance and double the width. The 1978 paper presents the values for a and b for their tested devices and shows that the values predicted by the formula are quite close to the values measured in the experiment (see Figure 4.22).

Later researchers refined the analysis to take into account different *directions* of movement. Depending on the ergonomics of the input device, there might be different muscles involved when moving horizontally rather than vertically, so the movement parameters might not be identical. For example, the "RollerMouse Red *Plus*" from Contour (Figure 4.23) uses a cylinder as the pointing device, which the user *slides* left and right for horizontal movement and *rolls* forward and backward for vertical movement. Thus, in terms from Section 4.12.2, it might be an *absolute* device horizontally (since the cylinder slides left and right within the bounded space) but *relative* vertically (since it rotates unencumbered). Alternatively, it may also be relative horizontally, but since it cannot clutch, it has to use constant speed

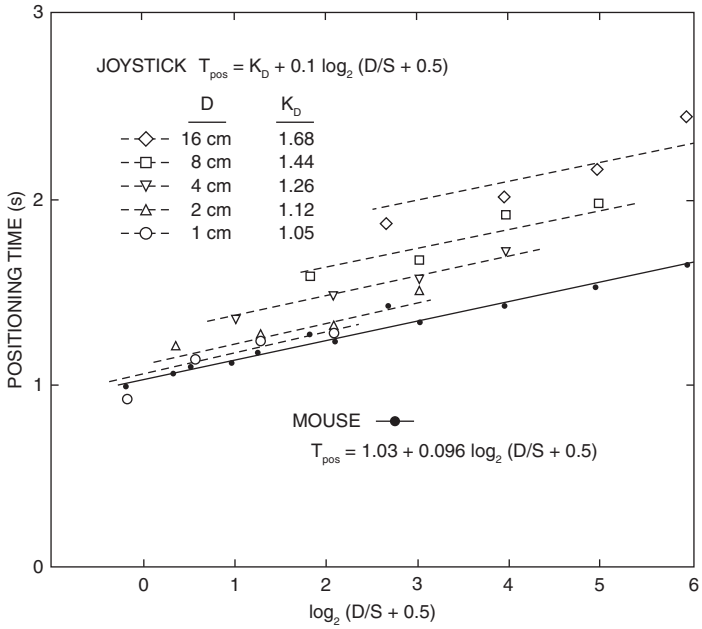
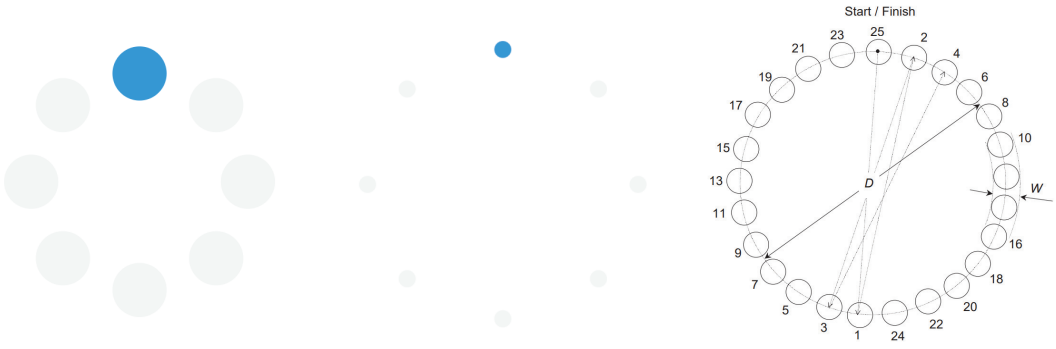


Figure 4.22 The values calculated for *a* and *b* for various devices result in curves that closely match the measured values. Source: Card et al. [1978].

Figure 4.23 RollerMouse Red *Plus* pointing device from Contour. The bumpy black cylinder (at arrow) is *slid* left and right and *rolled* forward and backward to move the pointer cursor. Source: https://contourdesign.com/collections/rollermouse/products/rollermouse-red-plus.

(rate-controlled) movement when the user hits the end. With respect to Equation (4.1) above, this device is likely to have different values for horizontal and vertical movement for the constants *a* and *b*. In other words, it may be faster to move the same distance in one direction versus another.



(a) (b) (c)

Figure 4.24 Two examples of the circular pointing device speed and accuracy test, with different size and distance of targets. The blue dot target moves across the circle. Source: (a) and (b) Author's screenshots. (c) Soukoreff and MacKenzie [2004].

Therefore, researchers developed the circular pointing device test, shown in Figure 4.24, where the user moves back and forth across the circle in all directions [Soukoreff and MacKenzie 2004]. The target dots can be various sizes and at varying distances, as in the original Fitts' law tests. This multi-dimensional version has been standardized as the ISO9241-9 standard (2002) [Soukoreff and MacKenzie 2004]. There are a number of online and offline versions for testing Fitts' law, including ours:

- http://simonwallner.at/ext/fitts/
- https://www.cs.cmu.edu/~bam/uicourse/05440inter2022/fitts-law-test/
- http://www.yorku.ca/mack/FittsLawSoftware/

4.15 Research About Pointing

There has been substantial research on how to make pointing to the screen faster, more accurate, and more comfortable. Section 4.14 already covered research on how to evaluate the speed and accuracy of pointing. In addition, there is a large amount of research on improving the speed and/or accuracy of existing pointing devices. For example, one line of work aims to "Beat Fitts' law"—for example, by making the active area of the cursor be a shape instead of a point. For example, the *bubble cursor* adjusts the size of the area to be a circle around the cursor point just big enough to point to exactly one object and was shown to be statistically significantly faster than using a regular cursor for selection [Grossman and Balakrishnan 2005]. Note that in Fitts' law terms this decreases both the distance to the target

one must move and increases the effective size of the target. A similar idea is provided by *drag-and-pop* [Baudisch et al. 2003], which detects the direction the cursor is moving and temporarily moves the targets that can be reached in that direction to be closer to the cursor so that they can be selected more easily. If the user selects something or moves in a different direction, they pop back to where they were. Most recent research focuses on pointing in 3D and augmented and virtual reality and is discussed in Section 12.5.

4.16 Summary of the Considerations for Pointing

Although pointing is fundamental to most interaction techniques and has been studied for computers for over 50 years, there is still room for improvement and for the invention of new pointing devices and techniques. Individual users also must select which input devices to use, both globally (which to buy) and often incrementally (which of the two to four input mechanisms—pointing stick, touchpad, touchscreen, attached mouse—should be used for the next interaction?). This section summarizes some of the considerations:

- Accuracy—possibly down to sub-pixel movements.
    - Continuous (every pixel) versus discrete (e.g., jump from item to item like with arrow keys).
    - Accuracy when buttons are pressed (side button moves a pen).
    - Jitter/noise in the sensor (like with magnetic pens).
- Speed for short and long movements.
- Relative versus absolute positioning:
    - Is there support for digitizing?
    - How much space (e.g., on a desk) is required?
    - What resolution and speed of movement is supported?
    - If relative, then which way does movement on the device move the cursor on the screen?
- Comfort for long-term use—fatigue of fingers (repetitive stress injuries) and arm (gorilla arm syndrome). Can users rest their hand on the desk, like with a mouse or touchpad?
- Feedback so the user knows where they are pointing (e.g., cursor vs. finger location) and the current mode (cursor picture).

- How many states—for example, is hover supported versus touching versus selecting? Also, does it measure pressure, which provides continuous instead of discrete values? How many associated buttons are there (e.g., mouse buttons, pen tip press)? Is there the possibility of modifiers, potentially from the keyboard or special buttons—like Shift-click?
- What do the states do by default (e.g., how the states are used)—click, press-and-hold, and so on?
- How many dimensions of input are supported (1D, 2D, 3D)? (See also Section 19.5.)
- Rate versus position controlled (for joystick, but also for edge of touchpads)?
- Does it involve an extra device (mouse, stylus) and, if so, where does it go?
- What is the time to initiate using the device (e.g., time to move hand to a mouse from the keyboard, time to pick up a stylus)?

4.17 Chapter Conclusion

This chapter has summarized the development of 2D pointing devices, showing the many ways that users can indicate a particular location to computers and other technology. The most-used technique for pointing today is to just tap with a finger on a touchscreen, but this still has disadvantages like low resolution and occlusion caused by the finger. Therefore, new techniques are still being researched along with new ways to use pointing devices with widgets on the screen and more advanced pointing devices—for example, for 3D and pointing in the real world (see also Chapter 12). The rest of this book discusses how pointing is used to support many different tasks.

CHAPTER 5

Scrolling and One-Dimensional Input

5.1 Introduction

Moving around to control what part of the content is visible is a fundamental interaction technique that has been needed since the beginning of writing. Our term for this process, *scrolling*, comes from the earliest known way to write down long texts—using a scroll of parchment, which dates back to at least 4,500 years (Figure 5.1). Scrolling has also been around since the beginning of display screens on computers since there has always been more content than would fit at a time on displays.

Scrolling is also called *panning*, which comes from cinematography, where it means to move (or *pan*) the camera so that the contents move while generally staying the same size. This is as opposed to *zooming*, where the contents get smaller or bigger, and *rotating*, where the orientation of the contents changes.

Scrollbars (Figure 1.10) are an on-screen widget that supports moving around in a long (and/or wide) document. They date back to at least the earliest explorations of GUIs in the early 1970s on the Xerox Alto (Section 2.4.3), and they continue to be used both for vertical and sometimes also horizontal scrolling.

There have also always been alternative ways to scroll—examples include special keys, knobs, the mouse scroll wheel (Figure 4.17(b)), and a *flicking* gesture. Although two-dimensional (2D) scrolling is occasionally supported (e.g., with the *hand* icon, Section 5.7), users must usually scroll each direction separately.

When other kinds of one-dimensional (1D) input are needed (e.g., for selecting a number between two ranges, such as between 1 and 100), you might use a scrollbar widget, but might instead use a more special-purpose widget like a *slider*. All of these are covered in this chapter. Note that the equivalent of scrolling for three dimension (3D) is covered in Chapter 12.

Figure 5.1 Scrolls of parchment date back over 4,500 years. Source: By "Boekrol Esther 18de eeuw uit een sefardische synagoge in Sevilla" (text from pic). Public Domain, https://commons.wikimedia.org/w/index.php?curid=3758954.

5.2 Auto-Scroll

It is also worth mentioning *auto-scroll*, which devices have also long supported. This is when the selection of text (Section 8.6.3.2) or graphics (Section 10.5) moves to an item that would otherwise be outside of the screen or window, then the system will typically scroll to make the item visible again. For example, if the text cursor is on the last visible line of text, and the user moves the cursor down one line, the system will scroll the text up one line, so the cursor is again visible. Similarly, when dragging an item with the pointing device, if the user moves it to the edge of the screen, the contents will start scrolling in the opposite direction, so it can be dropped in a place that was formerly off-screen. The Alternate Reality Kit system (Section 2.4.10) from 1986 used this as its primary way to scroll—when the cursor was moved to the edges of the window, the contents would start scrolling (even if nothing was being dragged) [Smith 1987].

A key design parameter for this is how fast to scroll. For keys, it is usually the auto-repeat rate (see Section 5.5), but for a pointing device, it may depend on how far the user moves the cursor away from the edge of the window. For example, moving just to the edge and holding there would usually cause a slow scroll,

whereas moving further past the edge would cause faster scrolling. This brings up the problem of *overshoot*—when the scrolling is happening so fast that the user misses the desired target and must move back in the other direction—which often happens when scrolling is too fast to see or when the user cannot react quickly enough if the target is seen.

Another issue with auto-scrolling is *aborting* (see also Section 1.8.3)—can the user stop the action while it is in progress? Interestingly, hitting the "Esc" keyboard key while auto-scrolling on a PC will usually *stop* the auto-scrolling, but it does *not* revert the contents back where they used to be, so the scroll itself is not undone or canceled.

5.3 Knobs

A knob is usually a 1D input device that can either be physical or virtual. Physical knobs were available on the TX-2 computer used by Sketchpad (Figure 2.8) in 1963, with the four knobs below the display being used to scroll and zoom horizontally and vertically. An important example of the use of a knob is the roller dial on the front or side of the older RIM Blackberry devices (starting about 1999, see Figure 3.23). Since these early devices did *not* have a touchscreen, this dial was the chief way that users would move around the content.

Another famous "knob" is the scroll wheel of the original iPod from 2001 (Figure 3.25), which physically turned (later iPods used virtual knobs, which was basically a circular touchpad, that measured the user's circular finger movement).

Today, we use physical knobs for scrolling as the scroll wheel of a mouse (Figure 4.17(b)), which was popularized by the Microsoft IntelliMouse in 1996, along with built-in support for the mouse wheel in Microsoft Office 97. A control panel setting specifies how many lines are moved by each increment of movement (called a "notch" by Windows 10) of the wheel (see Figure 5.2).

Knobs also appear in cars (Figure 4.5) and in other consumer electronics. For example, an early wearable device from 1992 with a head-mounted display used a separate knob worn on the hip as a controller that could be used without looking at it (see Figure 5.3) [Finger et al. 1996].

I used a virtual knob as a user-customizable on-screen widget in the Pebbles Shortcutter program for the Palm Pilot in 2001 [Myers 2001], where circular movement could be mapped to changes in zoom level, volume, or other 1D values (the knob is labeled "Zoom" in Figure 5.4(b)).

Knobs may be absolute or relative, in the same way as pointing devices (Section 4.12.2). A knob that supports absolute selection of the value has stops so it can only turn a certain amount, typically somewhat less than one revolution. This makes it easy for users to know when they have hit the minimum and maximum

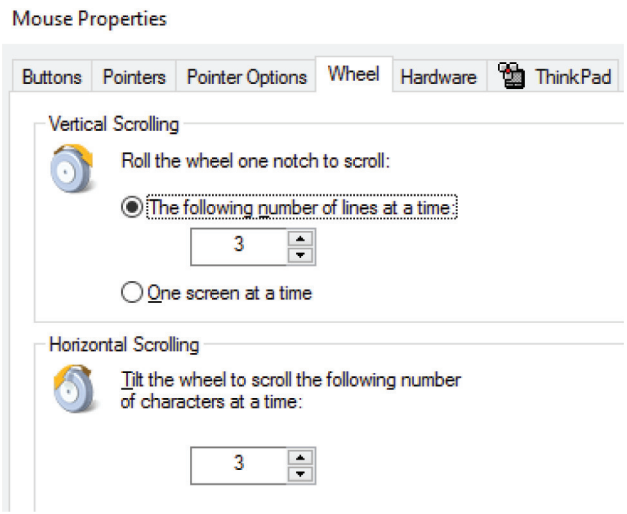


Figure 5.2 Scroll wheel properties in Windows 10, set to three lines per increment ("notch"). Source: Windows 10 screenshot, used with permission from Microsoft.

Figure 5.3 Early wearable system used a head-mounted display and a knob on the hip that can be operated without looking at it. Source: Finger et al. [1996].

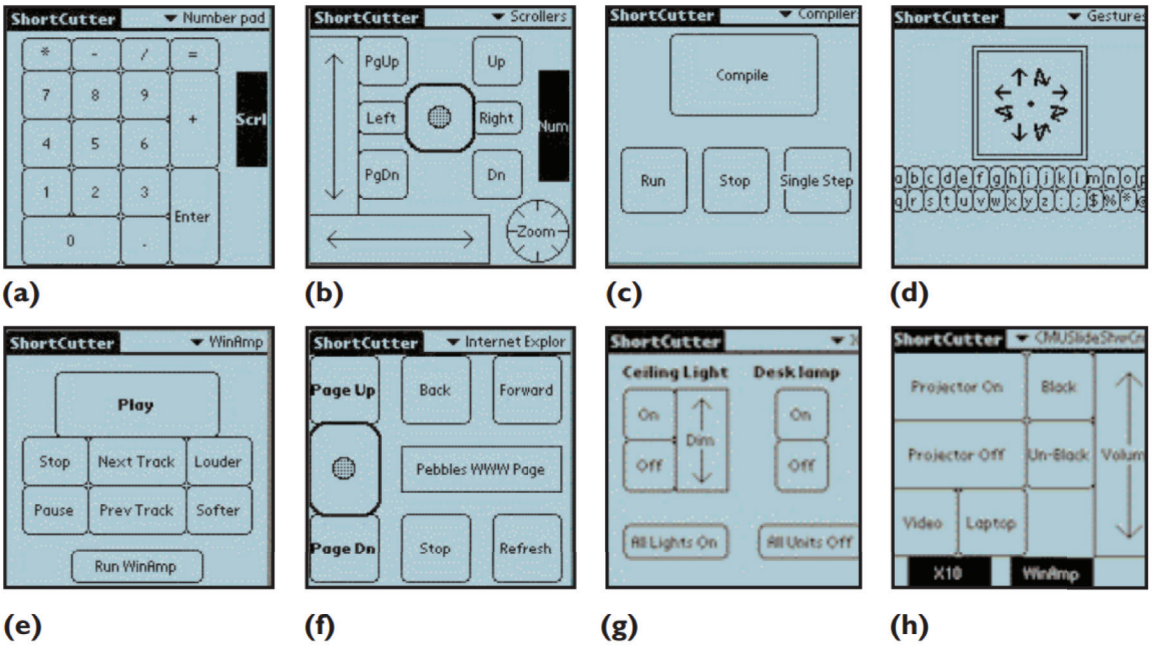


Figure 5.4 Panels created with the Pebbles Shortcutter program: (a) numeric keypad; (b) collection of scrollers and a knob; (c) buttons for controlling items in a set of compilers; (d) gesture pad and two rows of small buttons; (e) controller for the WinAmp PC program for playing music files; (f) panel for browsing via Internet Explorer; (g) panel that interfaces with X-10 to control room lights; and (h) panel that controls a video projector. Source: Myers [2001].

values. A relative knob generally rotates continuously in either direction, with each increment of movement mapped to an increment of the value. Like with pointing devices, this mapping does not need to be 1-to-1. For example, as explained in Section 3.17, the iPod scroll wheel was carefully designed so that if users spun faster, the contents would scroll even faster.

Another design choice for knobs is whether they "click" with each increment or move smoothly. Clicking makes it easier and more accurate for users to feel how many increments they have moved (e.g., to move exactly one increment) but make it impossible to move at higher resolution (e.g., half of an increment is not possible). Most scroll wheels on mice have clicks for each increment, called a *notch*. Another dimension is whether the knob can be flicked so it continues to move after the user releases it.

5.4 Physical One-Dimensional Sliders

There have been a few research and commercial approaches to providing linear 1D input from a physical slider. The classic examples are the sliders on an audio

console (also called a mixing console), which typically have one slider and a set of knobs for each input channel (see Figure 5.5). These sliders are obviously absolute, with a clear minimum and maximum value. It has also been common to use a narrow touchpad as a 1D slider, which may either be absolute or relative (in the same way that touchpads can be absolute or relative in 2D, Section 4.12.2). For example, before two-finger scrolling became commonplace with the iPhone, many laptop touchpads supported scrolling the content by dragging the finger along the right or bottom edges. As an example of research using 1D sliders, Bill Buxton and I showed [Buxton and Myers 1986] that people could use a 1D slider on a touchpad with their left hand, either in absolute or relative mode, *at the same time* as using a mouse-like pointer device in their right hand, even when they were not instructed to use both hands at the same time (see also Section 5.11.1).

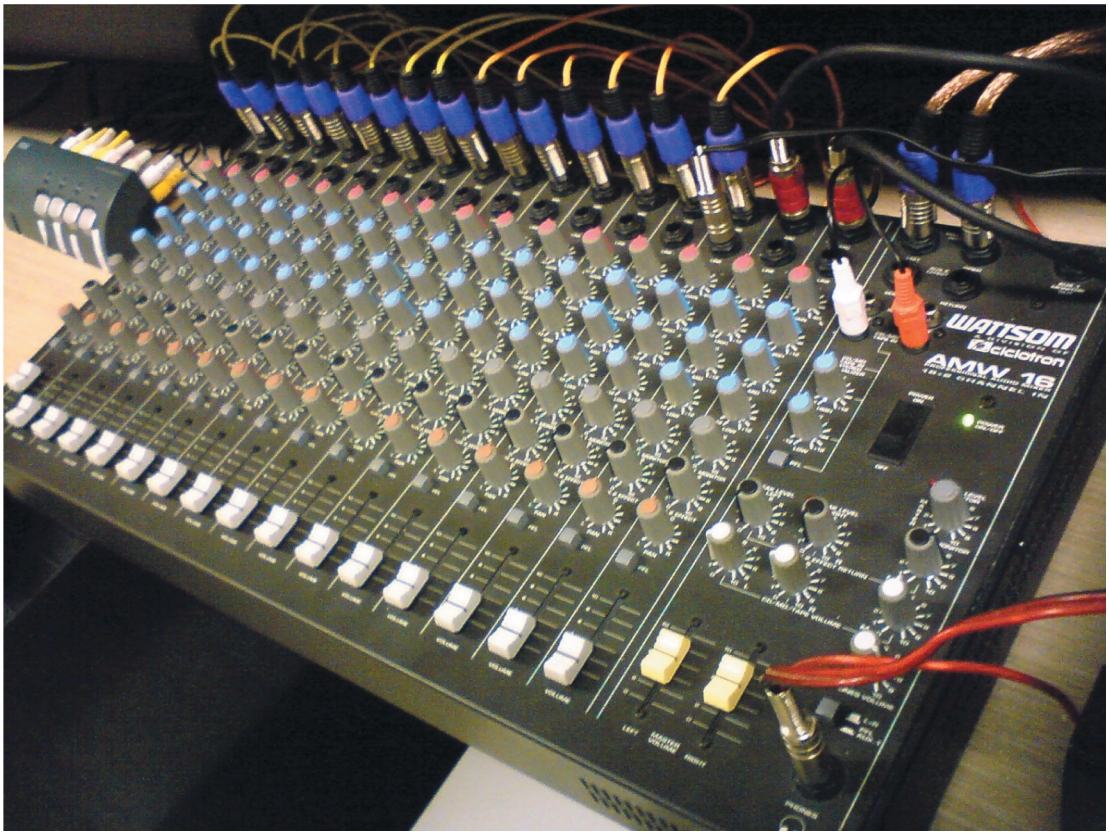


Figure 5.5 Audio mixing console with 1D sliders at the bottom and knobs above. Source: By Rodrigo César—Own work, Public Domain, https://commons.wikimedia.org/w/index.php?curid=4356127.

5.5 Keys and Physical Buttons

Users have always been able to move around the content using arrow or scroll keys. These are used not only on desktop computers and remote controls (see Figure 4.4) but also on mobile devices, such as the Palm Pilot up/down arrows (Figure 3.18(a)).

As mentioned in Section 5.2, sometimes these keys will move a selection, and when the selection gets to the edge, this will cause scrolling. Other times, the keys will directly cause scrolling, as shown in Figure 5.6 for (a) WordStar for DOS from 1978 and (b) the Page Up and Page Down keys for PC. These methods for scrolling continued to be supported even after GUIs and the mouse became commonplace.

All keyboard keys and some buttons support *auto-repeat*, which means that when one holds them down, they repeatedly send out their value. Typically, there is an initial delay before this starts, so the user can press and release them without auto-repeating by accident. For example, on Windows 10 the default delay is about ½ second. The second parameter is how fast the values are generated. The default on Windows 10 is approximately 30 characters per second. Together, these mean if you press the Page Down key (Figure 5.6(b)) and release it before 500ms passes, then you will get one Page Down event. If you continue to hold the Page Down key longer than 500ms, you will continue to get Page Down events about every 33ms (so you will get 30 per second) until you let the key up. The delay time and the repeat rate are both controlled by a setting available in the Windows Control

(a) (b)

Figure 5.6 (a) WordStar scrolling commands (circled in red) and soft key labels for the F1 to F10 keyboard keys at the bottom (1978). (b) Some labeled keys from a PC keyboard. Sources: (a) Wordstar, fair use, via Wikipedia, https://en.wikipedia.org/wiki/File:Wordstar_Screenshot.png. (b) Mysid, image was created with Inkscape, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=6053211.

Panel for the keyboard, and there is an equivalent mechanism for other operating systems.

Similarly to auto-scroll (Section 5.2), a problem with auto-repeat is overshoot—if the user passes the target while the key is held down. Then the user will typically have to go back the other way.

5.6 Scrollbars

Scrollbars (sometimes also written with a space: "scroll bars") are one of the oldest GUI widgets and date back at least to 1974. A *scrollbar* can be simply defined as an on-screen widget that allows users to scroll the contents. Scrollbars are interesting because they are a widget whose design and behavior have changed dramatically over the years, and scrollbars continue to change today, in contrast to many other interaction techniques that have stayed mostly the same.

Typically, a scrollbar incorporates *multiple* ways to scroll, to support the diverse needs users need to navigate around the contents. A typical scrollbar provides ways to move up and down incrementally, like one line at a time, as well as *paging*—moving by a full page or full screen. Scrollbars usually also support showing where in the document the user currently is, with an indicator that sometimes also shows what percentage of the file is being seen. The indicator will move from one end to the other of a section of the middle part of the scrollbar, sometimes called the *trough* (or *track*). Typically, this indicator, which is often called the *thumb* or *elevator*, can also be used to jump directly to a particular point in the file, by analogy with sticking one's thumb into a physical book to save a place or *thumbing* through a book as a way to skim it. In this mode, the height of the full scrollbar represents the full document (not just the part of the document that is displayed), and the thumb shows the user's position in the document.

For a programmer putting a scrollbar widget into a user interface, the scrollbar will typically be parameterized by whether this scrollbar is horizontal or vertical (or this may be selected by using a vertical-scroll widget vs. a horizontal widget), how large the document is (typically in characters or lines for a text document, or inches, millimeters, or pixels for images), how far through the document the user is (which may be expressed as a percentage or as a value between 0 and the content length, or may be a floating point number—e.g., between 0.0 and 1.0), what percentage of the document is currently being shown (for scrollbars where the thumb part changes size to show this value), how far the content should scroll when the user clicks an increment button (e.g., move one line or move 12 pixels), and how far to jump when scrolling by pages (when this is supported by the widget). Scrollbars can have many other parameters, as discussed in the rest of this chapter.

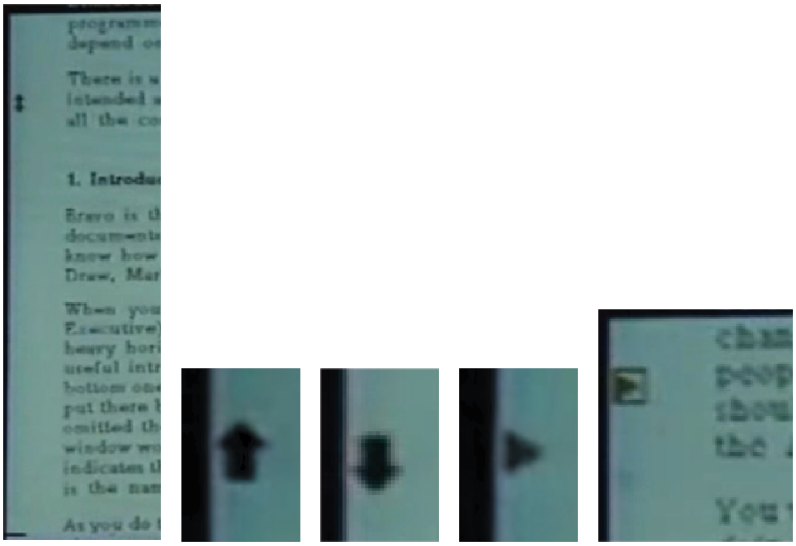
5.6.1 Bravo's Scrollbar, 1974

The Bravo text editor (Section 2.4.3.2) was one of the first GUI tools created by Xerox PARC. Since it allowed users to create long documents, they needed some way to scroll through them. Bravo used the *left* margin area of the document as the (invisible) scrollbar. When the mouse cursor was moved into the margin, the cursor shape changed to be an up–down arrow to show that scrolling is enabled (Figure 5.7(a)). Given the three-button mouse they were using, it seemed natural to use one mouse button to scroll up, another mouse button for scrolling down, and the third button to thumb. In order to help make these more learnable, when the user pressed down on a mouse button, the cursor changed to show what would happen, with the action only happening on mouse button release. This supported *aborting* since the cursor could be moved out of the scroll region before releasing to abort the operation.

The left mouse button was for scroll up (indicated by an up pointing arrow (Figure 5.7(b)), and it moved the line next to where the mouse cursor was to the top of the window, so the contents scrolled up. You might think that a symmetrical operation for scrolling down (Figure 5.7(c)) would move the line next to the cursor to the bottom of the window, but this is *not* what happened. Instead, when you pressed and released the right mouse button, the contents moved down so the top line of the document moved to be where the cursor was. This was designed to make the two operations reversible—hitting either the left or right mouse button, and then hitting the other button (without moving) would return the document to where it was. This form of scrolling supported both incremental movement (one or two lines at a time) and paging—if the cursor was next to the second line of the screen, the left and right mouse buttons would scroll one line up and down (and the user could control the increment by simply moving the mouse up and down from the top). If the cursor was at the bottom of the screen, the buttons would scroll up and down by pages. Note that this design did *not* support auto-repeat of scrolling since the action happened on mouse button *up*.

The middle mouse button supported thumbing. When the user pressed down on the middle button, the cursor changed to a triangle (Figure 5.7(d)), where the percentage of the way up and down the left edge of the screen represented that percentage of the way through the document. Releasing the button caused the contents to jump to that position in the document. Pressing the middle button again without moving would show an indicator (Figure 5.7(e)) of how far through the document you were [Xerox 1978, p. 34, Myers 1990b].

Interestingly, the other early PARC drawing and painting programs discussed above (Section 2.4.3.3) apparently did *not* support scrolling at all. Markup had a



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

Figure 5.7 (a) Bravo's scrollbar was the white region to the left of the text. When the pointer cursor was moved there, the cursor picture changed to an up–down arrow. (b) The left mouse button scrolled up, (c) the right button scrolled down, and (d) the middle button was thumb, (e) which also can be used to see where you are in the document. Source: Myers [1990b] around time codes 3:10–3:54.

way to create a new blank page and jump from page to page, but the Draw manual does not mention any way to change which content is shown on the screen [Xerox 1978].

5.6.2 Smalltalk's Scrollbar, 1977

As discussed in Section 2.4.3.1, the version of Smalltalk from about 1976–1977 supported multiple windows. The scrollbar for each window was hidden until the user moved the cursor into a window, at which point the scrollbar would pop up outside the left edge of the window. Thus, having a scrollbar visible was one way that the input focus window was indicated (see Section 11.4). Smalltalk experimented with having a single mouse button control most widgets and only the left button controlled the scrollbar. In the center of the scrollbar was a gray rectangle as the thumb (Figure 5.8(a)), which showed both what part of the contents and what percentage of the contents were visible in the window. When the pointer cursor was in the scrollbar and to the right of the thumb, the cursor changed to a half up arrow (Figure 5.8(a)) and the area to the left of the thumb was for scrolling down (Figure 5.8(b)). Clicking in these areas worked like in Bravo—it brought the line

(a) (b) (c)

Figure 5.8 (a) Smalltalk's scrollbar appeared when the cursor was in the window. The gray box in the center shows what part (both position and percent) of the contents is currently visible in the window. When the cursor was on the right side of the scrollbar, pressing the left mouse button scrolled up and (b) the left side scrolled down. (c) Pressing in the center would drag the thumb indicator up and down, with the outline box showing the original position. Source: Myers [1990b] around time codes 3:54–4:20.

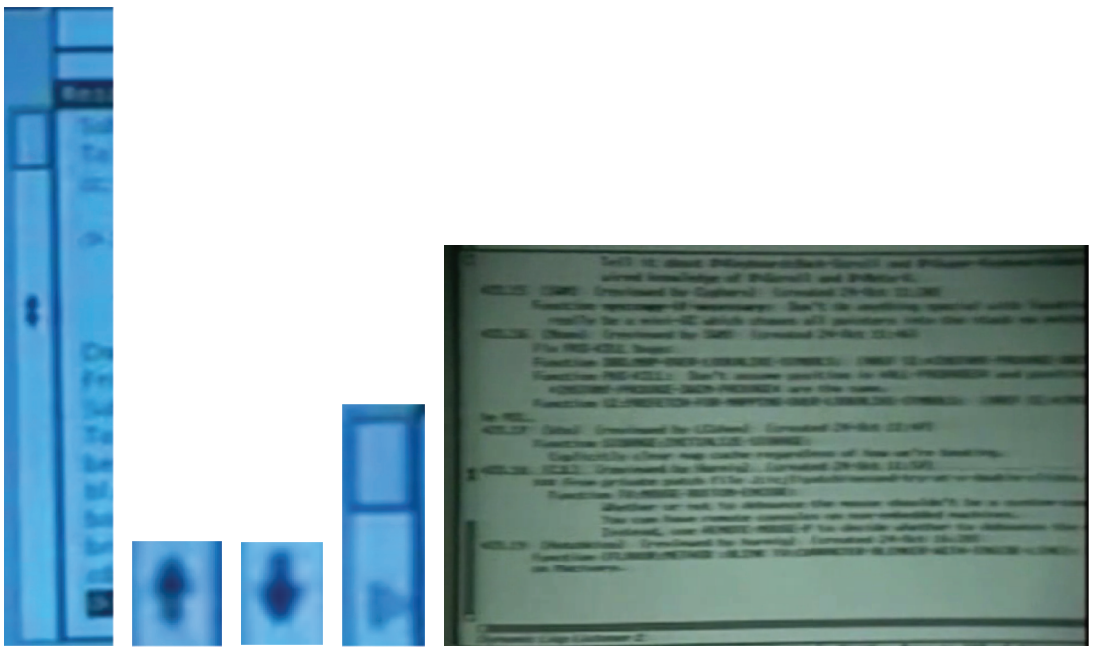
next to the cursor to the top or the top line next to the cursor. Pressing in the middle region (Figure 5.8(c)) allowed the thumb and corresponding window contents to be dragged smoothly up and down, with an outline box showing where it used to be Myers [1990b, at 3:54–4:20].

5.6.3 Interlisp-D, Symbolics Lisp Machine, XDE Scrollbars, and Andrew, 1980–1984

Interlisp-D (Section 2.4.3.4) developed a windowing system based on ideas from Smalltalk around 1980. Its scrollbar was an interesting combination of Bravo and Smalltalk. The first difference was that the scrollbar only appeared when the cursor was moved out of the left edge of the window (rather than when the mouse moved

into the window like Smalltalk). It had a thumb indicator showing the percentage of the document seen (Figure 5.9(a)), which was the full width of the scrollbar. Scrolling worked like Bravo, however, using the left mouse button for up, right for down, and middle for thumbing (Figure 5.9(b–d)). The Xerox Development Environment (XDE), also called Tajo (dating around 1981) and Cedar (1982, see Section 2.4.3.4), operated the same as Interlisp-D except the regions for scrolling were reserved sections on the left and bottom of windows rather than being popups.

As mentioned above (Section 2.2.2), inspired by the Alto, a number of commercial companies started to make personal workstations. Some of the first were produced by two companies creating special-purpose hardware for Lisp. The Symbolics Lisp machine had one of the first commercially released window managers around 1980, and it used scrollbars very much like Bravo and Interlisp-D, using the same three mouse buttons to scroll in a similar way. An interesting difference is that Symbolics had the current line move to the *bottom* of the window for scrolling up the document, whereas to scroll down, the current line moves to the top of



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

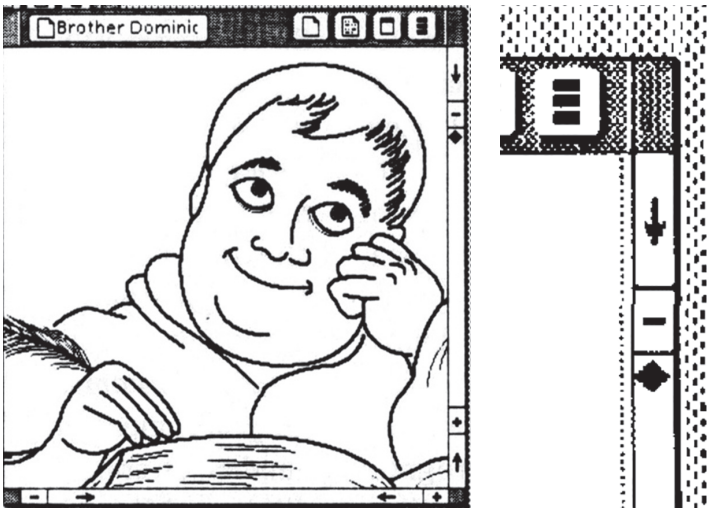
Figure 5.9 (a–d) Interlisp-D scrollbars worked like a combination of Smalltalk and Bravo, using all three mouse buttons. (e) Symbolics vertical and horizontal scrollbars. Sources: Myers [1990b] around time codes 4:20–5:20; 5:34.

the window like Bravo. As mentioned above, this seemed more symmetric but it made it harder to reverse the operation to go back to where one was. I believe that the Bravo design was more usable than Symbolics', which is illuminating about the differences of the philosophies of the two groups—Symbolics machines were designed for computer experts and so the company gave little thought to usability in general. Similarly, Symbolics scrolling happened on button *down* (rather than button up like in Interlisp-D), which meant that the user did not get feedback about what operation would happen until it was already finished. But this also allowed Symbolics to support auto-repeat of scrolling up and down if the mouse button was held down. Symbolics windows also provided horizontal scrollbars (Figure 5.9(e)).

Some other systems continued to use the three different mouse buttons to scroll, such as the Andrew system [Palay et al. 1988] from IBM and CMU (see Section 2.4.12) from around 1986. This system added small rectangles at the top and bottom that when clicked on would move the document directly to the top and bottom (so the user doesn't need to drag the thumb or hold down the mouse button a long time to get there). It also added an indication in the scrollbar to show where the selected portion of the text was (which was not necessarily the part that was showing on the screen).

5.6.4 Xerox Star (1981) and Viewpoint (1985)

The style of scrollbar that we are familiar with today dates back to the Xerox Star in 1981. As explained in Section 2.4.6, the Star had a two-button mouse, with the left button being the main one used and the right button for *extend*. Therefore, the scrollbar needed to work with a single mouse button. The Star wanted the targets to be more apparent, so explicit on-screen buttons were provided (see Star overall picture in Figure 2.14 and details in Figure 5.10). Note that the scrollbars on the Star were moved to be on the *right* side of the window to reduce visual clutter. Star's scrollbars added the arrows at the top and bottom. Note that the top arrow pointed *down* (Figure 5.10(a)) and the bottom arrow pointed *up*—we will see that Apple switched this (Figure 5.11). Clicking on the top arrow caused the contents to scroll down, which was the direction that the arrow at the top pointed. This moved the view toward the *top* of the window and consequently the thumb, shown by the diamond in Figure 5.10(b), moved toward the top of the scrollbar. The minus button at the top and the plus button at the bottom of the scrollbars (next to the arrows) moved by pages. These buttons operated on mouse down, and they auto-repeated. Pressing in the thumbing region dragged the thumb to that part of the document (rather than causing paging as on the original Macintosh and later scrollbars). A diamond appeared in the scrollbar to indicate what part of the document was currently showing in the window, but not what percentage (Figure 5.10(b)). Oddly, for



(a) (b)

Figure 5.10 Details of Xerox Star (a) window and (b) scrollbar, which also shows the menu or "hamburger" icon (see also Figure 7.2(a)). Sources: Johnson et al. [1989].

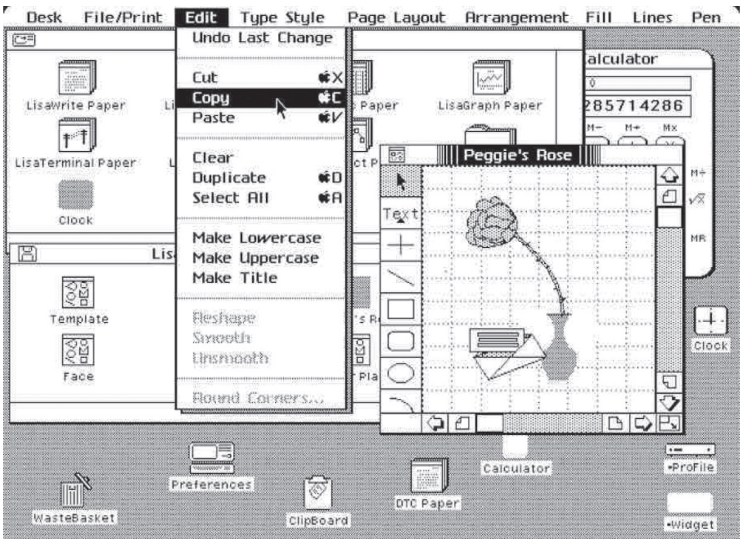


Figure 5.11 Apple Lisa screen from 1983. LisaDraw is shown in the window at the center-right with the title "Peggie's Rose." Its scroll bar is at the right, and its palette on the left has the top arrow selected. The menu bar at the top has "Copy" selected from the "Edit" menu and contains various available and grayed out menu items. Source: Courtesy of David T. Craig.

horizontal scrolling, Star only had two arrow buttons and the plus and minus, and not the thumb indicator or trough (Figure 5.10(a)).

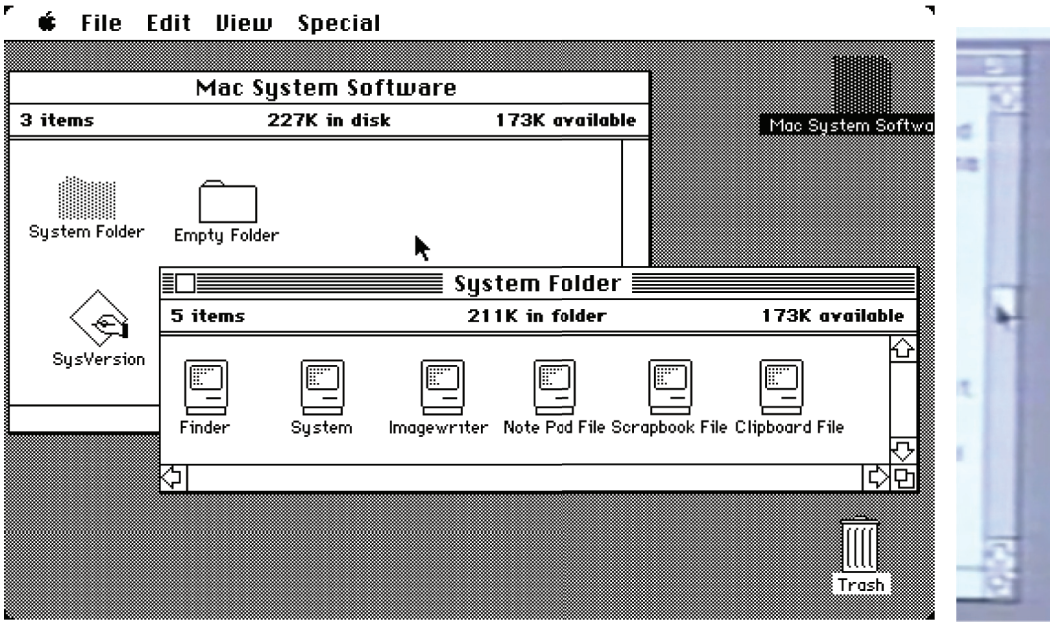
In 1985, Xerox released a new version of the system called Viewpoint, which had a few interesting changes to the scrollbars. First, when pressing and dragging the thumb, the cursor was allowed to go outside the region (as discussed below for the Macintosh released the year before). Also, while dragging the thumb, Viewpoint changed the cursor picture to show the page number of the part of the document to be scrolled to.

5.6.5 Apple Lisa (1983) and Macintosh (1984) Scrollbars

The next important designs for scrollbars came from the Apple Lisa in 1983 and then Macintosh in 1984 (Section 2.4.7). On both, the scrollbars were shown on a window when it had the keyboard focus, as in Smalltalk, as one signal of which window was active. The Lisa used two sets of icons at the top and bottom (Figure 5.11), like the Star. The outer arrow icons were for incremental scrolling. Note that they pointed *out*, indicating the direction that the thumb would move (and the opposite direction from how the text would move), which was the direction of all subsequent scrolling arrows. Next to the arrows were the page up and page down buttons (Figure 5.11). All of these buttons operated on first press and then would auto-repeat if the mouse button was held down. The Lisa and the original Macintosh used a fixed-size thumb indicator, which was a white rectangle on a gray background. Bill Atkinson said a fixed size was more understandable than a proportional indicator for short documents, and that they were moved to the right and bottom to make room for the window resize icon at the bottom right (Section 28.2). The Lisa and Macintosh used the same kind of scrollbar for horizontal scrolling.

The original Macintosh modified the design of the scrollbars somewhat. The main difference was the removal of the paging buttons as there was often no room for the thumb to move since users often had tiny windows on the small Macintosh screens (Figure 5.12). To scroll by pages, the user had to know to click in the gray region above or below the thumb indicator (which also would auto-repeat), but there was no affordance to help users know about this behavior. Pressing on the thumb itself and dragging it would jump to that percentage of the way through the document.

An interesting design decision about the Macintosh paging behavior was what to do when the indicator got to where the cursor was—imagine putting the cursor halfway between the thumb and the arrow icon, and pressing down and holding. This would cause the thumb to repeatedly move toward the cursor, eventually potentially moving *across* where the cursor was, which might cause the document to start scrolling the other way, which would be bad! Therefore, systems either stop scrolling (like the Macintosh and Windows) when the thumb gets to



(a) (b)

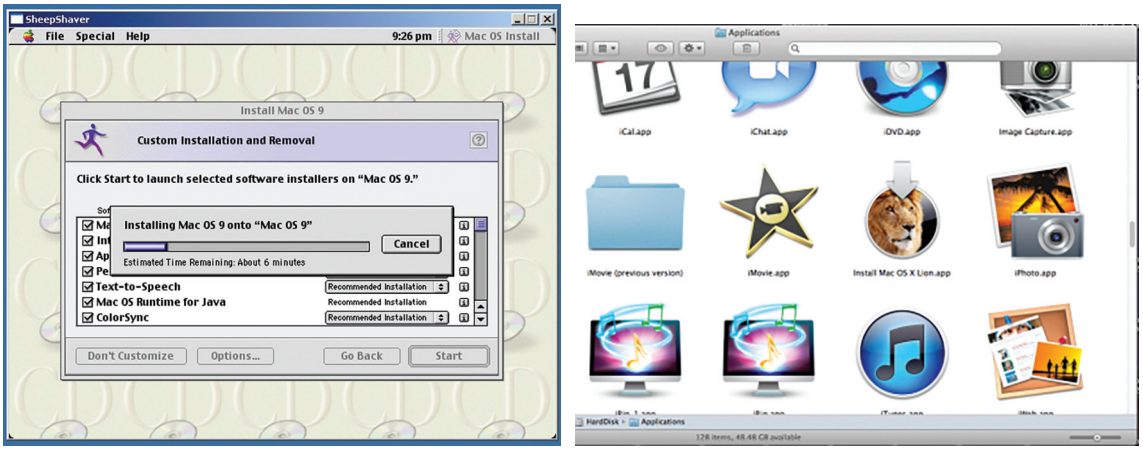
Figure 5.12 (a) Apple Macintosh screen. The System Folder window has the focus, so it shows the title bar decorations and the scroll arrows, but the contents fit into the window, so the gray thumb is not shown. (b) View of scrollbar from the video showing when content does not fit. Sources: (a) Apple Inc., fair use, https://en.wikipedia.org/w/index.php?curid=1016513. (b) Myers [1990b] around time code 8:11.

where the cursor is or else they just continue moving in the same direction, either moving the cursor (like OpenLook—Section 5.6.6) or basically ignoring the current relationship of the thumb to the cursor.

The user could abort scrolling while dragging the thumb by moving the cursor far out of the scrollbar before releasing the mouse button. Since it was difficult to move the mouse while staying in the narrow scrollbar region, the drag area extended about one inch on both sides of the scrollbar. Moving the cursor back inside this active region while continuing to press down resumed the scrolling.

The Macintosh displayed the arrows for windows that had the focus but did not show the gray elevator and thumb if the contents fit into the window, so no scrolling was needed (Figure 5.12).

When Steve Jobs rejoined Apple in 1997, he brought back with him the software from NeXT, including the look and feel of the NeXT scrollbars (see Figure 5.14(c)). Consequently, the scrollbar for Mac OS 9.0 in 1999 moved both of the increment buttons to the bottom, like NeXT had (Figure 5.13(a)). After the Apple iPhone came



(a) (b)

Figure 5.13 (a) Mac OS 9.0 in 1999 moved the scrollbar increment buttons to the bottom, based on the NeXT design. (b) Mac OS X 10.7 Lion from 2011 eliminated the increment buttons, based on the iPhone design. Sources: (a) Screenshot, Mac OS 9.0. (b) Screenshot, Mac OS X 10.7 Lion, 2011.

out in 2007, Apple started providing two-finger scrolling on its laptop touchpads (see Section 5.9), so the increment buttons became less important—users performed short scrolls using the two-finger swipe gesture, so the scrollbar lost the increment arrows altogether with Mac OS X 10.7 Lion from 2011 (Figure 5.13(b)).

Another interesting design change on the Macintosh was to make two-finger scrolling work on *any* window that the cursor was in, not just the front (focus) window (Section 11.4), so scrollbars were shown on *all* windows and no longer served as one of the signals to show which window had the focus (as of about OS X 10.8.0 Mountain Lion in 2012). And then scrollbars began to not be shown at all on any window except *while* the user was scrolling (by about OS X 10.9 Mavericks in 2013).

5.6.6 Windows, OpenLook, and NeXT Scrollbars

Most of the systems that were developed subsequent to 1984 and through the 1990s (including Microsoft Windows (1985) and Presentation Manager (1988), DecWindows (1989), and HP's Motif (1989)) used scrollbars that were quite similar in behavior to the Macintosh's, sometimes with small changes such as making the thumb be proportional instead of fixed size. An interesting difference was adopted by the OpenLook system from 1988 (discussed in Section 2.4.12). To make a direct analogy with a real elevator, the background of the scrollbar looked like a cable, and the up–down increment buttons were *part of* the thumb, and moved up and down with it (Figure 5.14(a)). You may wonder what happens when you auto-repeated by

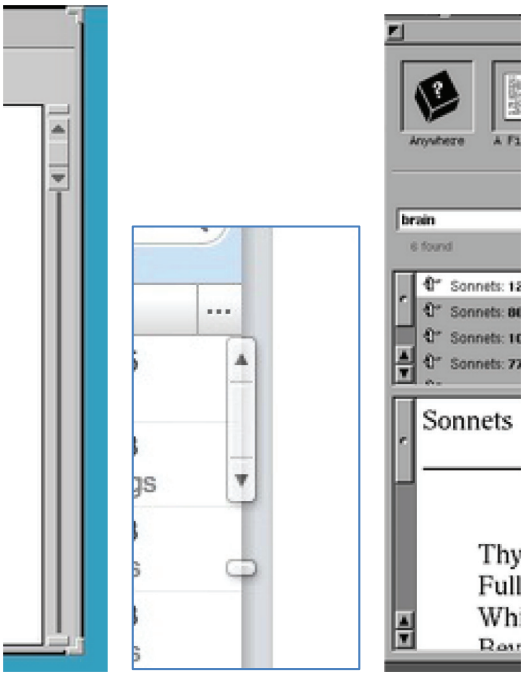


Figure 5.14 (a) OpenLook (1988) put the increment arrows on the thumb, (b) as did Google Wave (2010). (c) NeXT (1989) moved the scrollbar back to the left and both arrow buttons to the bottom. Sources: Screenshots of (a) Openlook, (b) GoogleWave, (c) NeXT.

holding down the mouse button over an arrow? OpenLook assumed it was always using an *indirect* pointing device (4.12.1) like a mouse, so it just moved the cursor along with the thumb. Similarly, if the user auto-repeated paging by pressing above or below the thumb, when the thumb got to the cursor position OpenLook simply moved the cursor along with the thumb. Note that this design wouldn't work if you had a direct pointing device like a touchscreen—the system couldn't move your finger like it could move the cursor! This design was also adopted by Google Wave (2010) (Figure 5.14(b)).

The NeXT user interface from 1989 moved the scrollbars back to the *left* side of the window and grouped both increment arrows at the bottom of the scrollbar (Figure 5.14(c)). This made it easier for users to switch which direction they were scrolling—for example, if they overshoot. The NeXT also had a different way to move by pages, which was triggered when the user clicked on an increment arrow while holding down the Alt keyboard key. Therefore, clicking in the trough of the scrollbar jumped the thumb to that position (rather than triggering moving

by pages). Another innovation was that dragging the thumb while the Alt key was held down scrolled more slowly, which was useful for large documents and where regular dragging would not provide fine enough control. (Note that using the Alt key when pressing the mouse button made the thumb scroll more *slowly* but it made the increment buttons scroll more *quickly*—an interesting inconsistency that I have not seen picked up by any subsequent system, including ones like the Silicon Graphics window manager from 1989 that copied other aspects of NeXT's scrollbar design.)

5.6.7 Mobile Device Scrollbars

Even though mobile devices such as personal digital assistants (PDAs, see Section 3.2.7) and pen computers (Section 3.2.4) had significantly different form factors than desktop computers, they still used essentially the same kinds of scrollbars. For example, the PenPoint's scrollbars (1991, see Figure 3.9(a)) had arrows at the end and a proportional thumb that behaved the same as the Macintosh's. The original Palm (1996, Section 3.13) provided up–down scroll buttons, as discussed above, but also had an on-screen scrollbar for long documents that the user could tap with the stylus, including the Macintosh-like scrolling behaviors and auto-repeat (Figure 5.15). Not surprisingly, Windows CE and PocketPC scrollbars worked like regular Windows.

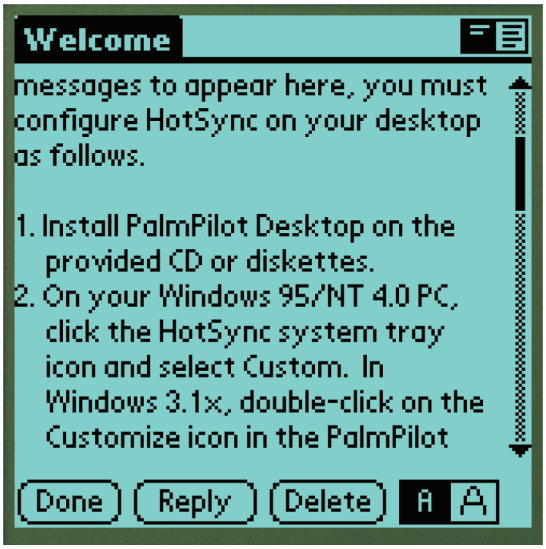
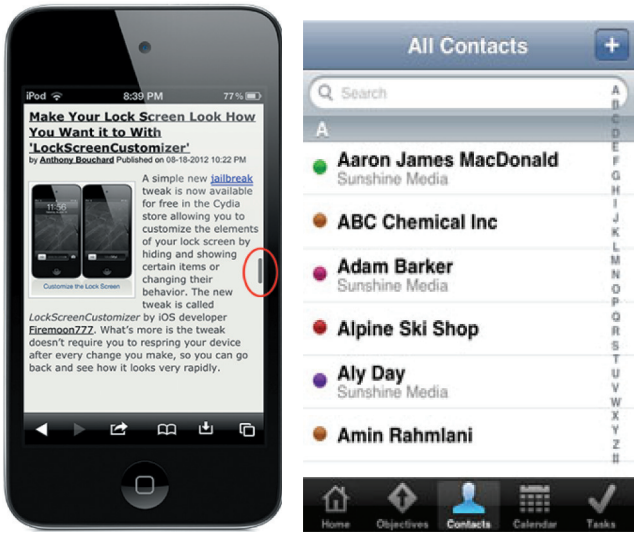


Figure 5.15 Palm scrollbar. Source: Author screenshot.



(a) (b)

Figure 5.16 (a) Main iPhone scrollbar (highlighted with a red ellipse) is just for feedback and is not clickable. (b) Special scrollbar for alphabetical lists. Sources: screenshots.

5.6.8 iPhone Scrollbars

The next major commercial innovation in scrollbars was the release of the iPhone (2007, Section 3.18). As discussed in Section 5.9 below, the primary way to scroll on the iPhone was to flick or drag with one or two fingers. Consequently, the main "scrollbar" provided was purely for feedback—it could not be tapped and it only appeared during scrolling (Figure 5.16(a)). The iPhone also introduced a special scrollbar for alphabetical lists that *could* be tapped on—where the full height of the screen represented the entire list, letters were displayed to the percentage of the way through the alphabet, and tapping on a letter jumped to that part of the content (Figure 5.16(b)). This was similar to the 1994 *AlphaSlider* idea [Ahlberg and Shneiderman 1994], which distributed the letters below a horizontal slider and explored many interactions to make it more efficient to select positions in long lists of alphabetical content. See also Section 5.9 for a discussion of other iPhone scrolling mechanisms.

5.6.9 Variations on Scrollbars

5.6.9.1 Showing Other Information in the Bar

It is now quite common to show other information in or next to the scrollbars to show the user where important items are in the file. For example, when one does a search for a string in a page in the Chrome browser, the scrollbar indicates all the

locations where the string appears (Figure 5.17(a)), and scrollbars for code editors similarly show all the locations of compile errors (Figure 5.17(b)). Note that in the first case the indicators are *in* the scrollbars, and in the second case they are next to it.

This idea has a long history, and many different kinds of information have been indicated in the scrollbar. As mentioned in Section 5.6.3, the Andrew system from around 1986 included an indicator in the scrollbar to show where the text selection was [Palay et al. 1988]. Value bars from 1992 showed various properties of the contents alongside the scrollbars [Chimera 1992]. The Seesoft system from 1992 showed a representation of the entire file using a tiny vertical strip [Eick and Steffen 1992], and the mural bar from McCrickard and Catrambone [1999] adapted this for use as the scrollbar (see Figure 5.17(c)). Other research has applied the same

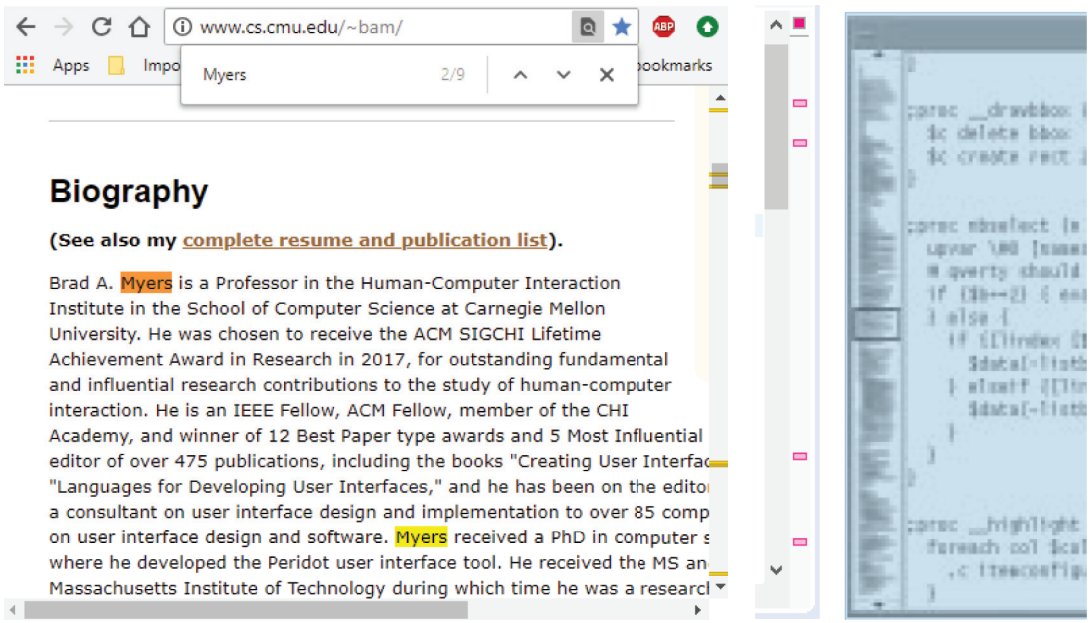


Figure 5.17 (a) The Chrome browser, version 66.0.3359.117 for Windows, displays orange lines in the scrollbar showing where the search term (here "Myers") appears in the document. (b) Similarly, the Eclipse code editor, version: Neon.3 Release (4.6.3), displays red rectangles next to the scrollbar to show where there are compile errors in the current file. (c) Shows a representation of the entire file as the scroll bar, with a box as the part currently viewed. Sources: (a) and (b) Screen captures, May 2018. (c) Excerpt from Fig. 3 of McCrickard and Catrambone [1999].

idea to sliders [Eick 1994], and it is now used in many code editors such as Visual Studio Code.

5.6.9.2 Auto-Hide

The scrollbars in Microsoft Word 2013 and later versions (including Office 365) disappear (*auto-hide*) if the user does not move the cursor for a while, even for the window that is in focus. Unlike the popup scrollbars in Smalltalk and other early systems, this does not save any space since the area where the scrollbar goes is still reserved. I think this is a really bad design since you can't tell what percentage of the way through the document you are without frequently wiggling the mouse.

5.6.9.3 Infinite Scrolling

Infinite scrolling, which is also called *endless scrolling*, is an interaction technique where when the user scrolls down to the bottom of the screen more content is loaded at the end of the current content, leaving the current location in view, so the user is no longer viewing the bottom. This technique became popular with Twitter and Facebook in the mid-2000s since they typically have too much content (possibly essentially unlimited content) to download it all. This is an alternative to chunking the content into pages, as provided by Google and other search engines and news outlets. Although this technique is claimed to provide a better experience for users in some situations [Ahuvia 2013], there are many disadvantages [Ahuvia 2013, Loranger 2014]:

- When you scroll toward the bottom of the contents, the thumb indicator goes to the bottom of the scrollbar but then jumps up a small amount when new content is loaded, so the user cannot really tell what percentage of the way through they are.
- Searching within the page does not work because the browser's search mechanism can only search the content that has already been loaded.
- Web search engines like Google and Bing do not index content beyond the first page. When my department switched to using an infinite scroll for the people here, only those whose names started with "A" through "D" could be found using Google, so we had to abandon using the infinite scrolling design on the people pages.
- Users cannot easily go back to known earlier places in the content using the scrollbar since the percentage of the way through the content is constantly changing.

- If the user knows that the target content is far down, it is tedious to repeatedly scroll to the bottom instead of directly jumping to a specific location.
- Psychologically, endless scrolling can be frustrating and can hurt the user experience as well. “For task-driven activities, infinite scrolling can feel like drowning in an information abyss with no end in sight.” [Loranger 2014].

5.7 Using a “Hand” to Scroll

Rather than (or in addition to) using a scrollbar, some systems provide the ability for users to *directly drag* the contents of the window around using the mouse (see Section 5.9.2 for dragging with a finger). An early example of this was the original Apple MacPaint hand tool from 1984. Note that MacPaint 1.0 did not have scrollbars (Figure 5.18). Instead, the user would select the hand icon (first column, second icon). Then, the user pressed down the mouse button at some location in the picture and dragged while holding the button down, and the picture moved with the cursor until the mouse button was released. This was useful for moving small distances but would be tedious for scrolling a long way, but that was not relevant to MacPaint since it was designed for small pictures given the original Macintosh’s small screen and limited memory to hold pictures. Note that another advantage of this interaction technique is that it supports scrolling in any direction

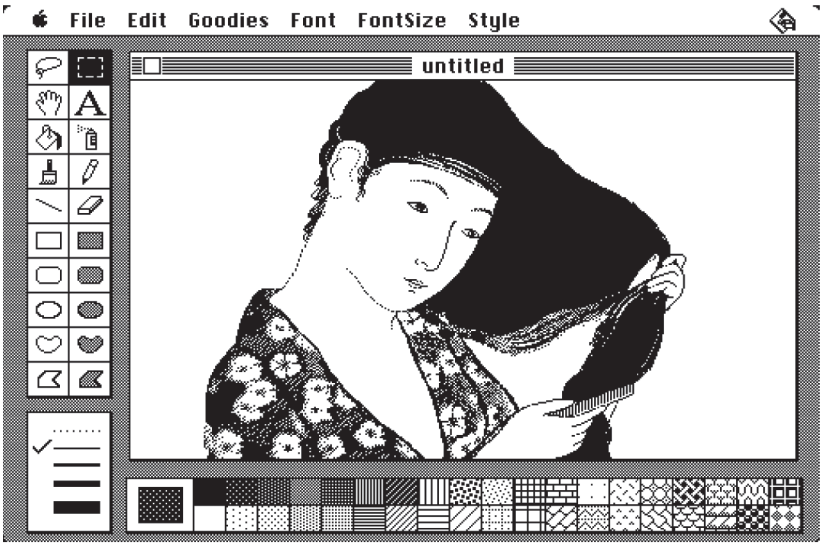


Figure 5.18 A picture in MacPaint 1.0, drawn by Susan Kare. Source: MacPaint, Fair use, https://en.wikipedia.org/w/index.php?curid=429029.

in two dimensions, unlike scrollbars that only allow scrolling in one direction at a time.

The hand icon continues to be a popular option for scrolling. For example, it has been available in Adobe Acrobat as a scrolling option along with regular scrollbars (Figure 5.19(a)). The mouse cursor uses a closed hand while dragging the contents, and an open hand while just moving the pointer while the button is not pressed (Figure 5.19(b)). Many other Adobe programs (such as Photoshop, Illustrator, etc.) also support the hand scrolling option. This hand tool in Adobe products also supports zooming and other picture viewing operations using various option keys while pressing the mouse button. Google Maps and other modern map programs make this kind of scrolling the default operation for the mouse button, and the one- and two-finger scrolling on smartphones (covered below in Section 5.9.2) is similar.

Another related concept is to provide a zoomed-out overview of the entire 2D space with a rectangle to show the portion currently being viewed in the main, detail view, and to allow the user to drag this rectangle around to change which part is shown the main view. An early example of this was in the Spatial Data Management System (SDMS) [Herot et al. 1980] that provided the overview map on a different TV monitor (see Sections 2.4.5 and 11.11.1.2).

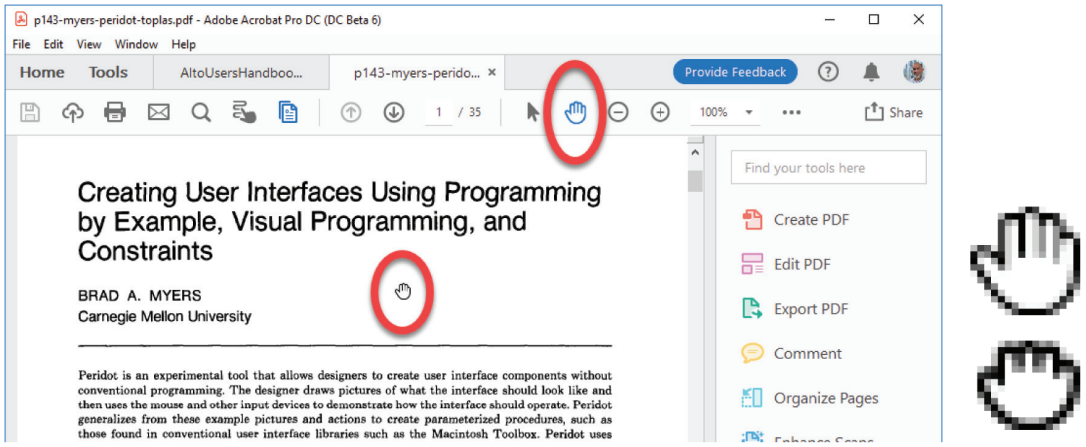


Figure 5.19 (a) Adobe® Acrobat® Pro DC (Beta 6), with hand scrolling cursor and icon highlighted. (b) Close up of hand cursor open and closed. Source: (a) and (b) Adobe product screenshot reprinted with permission from Adobe.

5.8 Rate-Controlled Scrolling

Another long-standing way to scroll documents is called *rate-controlled scrolling*. This is where the input device or widget controls the direction and speed at which the document scrolls rather than controlling the position directly. This has been the main way that the IBM TrackPoint supports scrolling. As shown in Figure 4.10, there are three buttons on the laptop below the pointing stick. If the user holds the *center* button down, then pulling or pushing on the pointing stick causes the document to scroll in that direction, with a speed proportional to how hard you press. For example, pushing the stick up a tiny amount causes the document to scroll slowly toward the top of the document (which means the contents move *down*). The pointing stick supports either vertical or horizontal scrolling, depending on which way the stick is pushed. Most implementations do not support scrolling diagonally since this is too hard to control. The mouse cursor changes picture to indicate the direction of scrolling (Figure 5.20).

Microsoft Windows operating systems also support rate-controlled scrolling using the scroll wheel on a mouse (Figure 4.17(b)). If you press in on the scroll wheel, it will click and the mouse cursor will change to show scrolling mode, and then moving the mouse while holding down on the scroll wheel (note: *not* rotating the scroll wheel) will cause the contents to scroll up or down at a speed depending on how far you move away from the spot where you pressed down.

I provided a virtual rate-controlled virtual joystick scrolling on a Palm device (Figure 5.4(b) in the center and Figure 5.4(f) at the left-center) in 1999, but a user study showed that this was 48% slower than using a conventional mouse to scroll [Myers et al. 2000b].

A key problem with rate-controlled scrolling is overshoot—when scrolling quickly, it is quite hard to stop when the target is reached. Sometimes, the contents are moving so fast that it is hard to even see the target go by, and even if the user sees the target, the normal human reaction speed means that users cannot stop exactly when it is reached. And switching directions with a rate-controlled device often results in overshooting in the opposite direction as well. These errors

Figure 5.20 The mouse cursor changes to show that vertical or horizontal rate-controlled scrolling is engaged. Source: Screen captures from May 2018.

and avoidance strategies (like slowing down) often mean that this mechanism for scrolling is quite slow.

5.9 Gestures for Scrolling

5.9.1 Flick-to-Scroll

As discussed in Section 3.5, the Go pen-computing tablet from 1991 supported making a *flick gesture* to cause scrolling (Figure 3.9(b)). This has also been supported by many subsequent systems, including modern smartphones. The content always moves in the direction of the flick—that is, putting a finger or stylus on the screen and flicking it up will also move the contents up (moving toward the *end* of the document). Typically, the content will move at a speed proportional to the speed of the flick, so users can flick faster if they want to scroll a further distance. The contents slow down and stop on their own, as if there was momentum and friction. In modern systems, the user can keep the contents moving and increase or decrease the speed of movement of the contents by flicking again while it is still moving.

A key feature of flick-to-scroll is *stopping* the scrolling by touching the screen with the finger or stylus. If the user sees the target content going by, the content can be stopped instantly by touching the screen. Typically, the system will avoid having this touch activate items on the screen. For example, if the content is moving as a result of a flick, and the user happens to stop the scrolling by touching on a hyperlink on the screen, that hyperlink will *not* be activated unless the user lifts the finger and puts it back onto the hyperlink.

Apple has patents on aspects of how this works, and at least one part—what happens when you scroll to either end of the content—has been upheld in multiple patent cases (I was involved in helping Apple overcome some challenges to their patents). US Patent 7,469,381 describes that "a displayed portion of the list of items may appear to bounce off of a boundary of the window in the touch-sensitive display when a beginning or an end of the list of items is reached" [Ording 2008, 21:27–30]. Figure 5.21(a) and (b) shows how this looks in the patent and also on the iPhone, where it seems like the content is attached with a rubber band so it doesn't move very far, and it bounces back. This patent was submitted in 2007, right before the first iPhone was released that same year. Android devices are not allowed to use this kind of feedback due to these patents, so some have displayed a translucent ellipse overlaying the content when you scroll past the end (Figure 5.21(c)). This is an example of when patents on interaction techniques can differentiate products and be worth a lot of money. This bounce at the end of the content is also what enabled the pull-to-refresh interaction, described in Section 32.2.

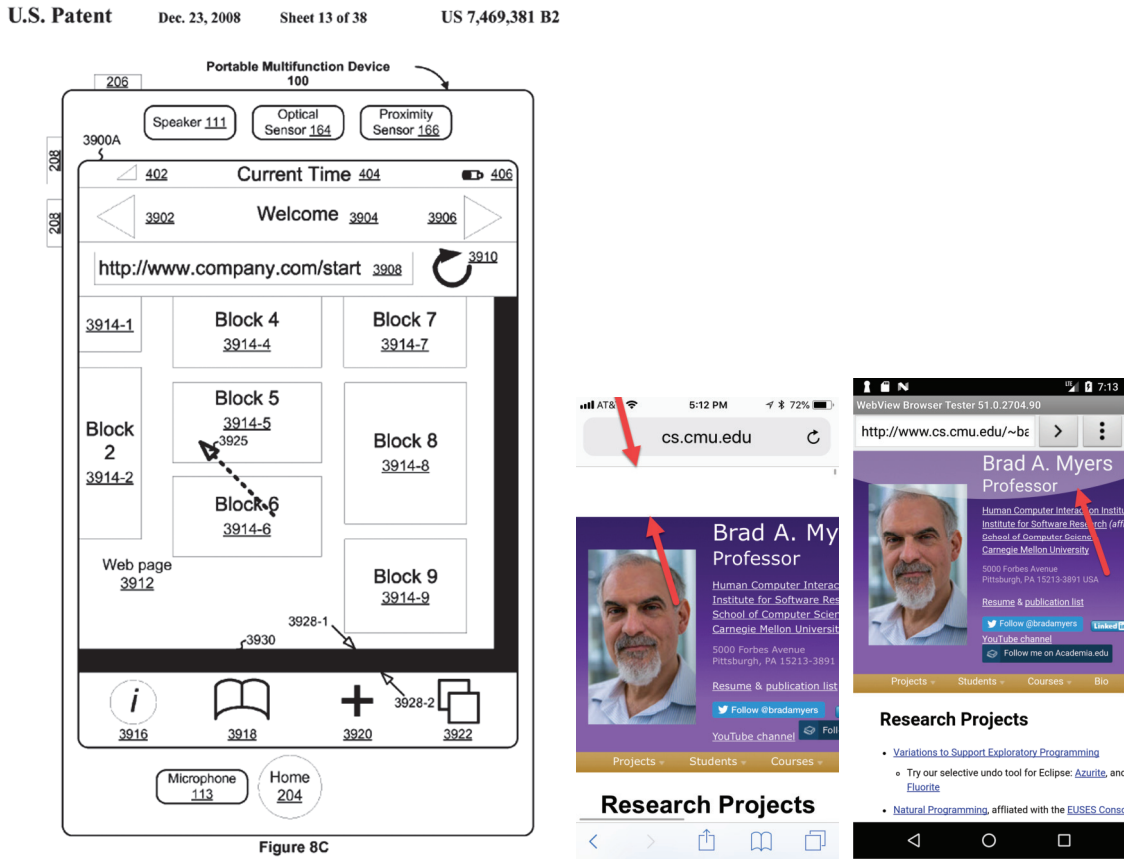


Figure 5.21 (a) US Patent 7,469,381 Fig. 8C shows that when the user pulls in the direction of the arrow labeled 3925, the content moves a shorter distance, between 3928-1 and 3928-2, and will bounce back when the user lifts the finger. (b) The iPhone operates similarly—see the space between the two red arrows as I pull down on my web page in Safari. (c) The Android default browser displayed a transparent elliptical overlay instead. Sources: (a) US Patent 7,469,381 Fig. 8C. (b) and (c) Screen captures from May 2018.

5.9.2 One- and Two-Finger Scrolling

Modern systems such as the iPhone also provide one-finger drag in addition to flick-scrolling. This means that if the user puts a finger on the screen and moves up or down, the contents are scrolled with the finger. The system must differentiate between scrolling and all the other things putting a finger on the screen might mean, such as pressing on a widget like a link or a button, starting to edit text, moving the text cursor location, popping up a context menu or dialog (e.g., for

long-press or "3D press"), moving a graphical object around, and so forth. Typically, one-finger scrolling is enabled whenever the user touches and immediately moves while keeping the finger on the screen, in any situation *except* where dragging of individual graphical objects is allowed (in which case moving the finger causes just that object to move, rather than scrolling the screen). One-finger move is different from flick-to-scroll (Section 5.9.1) in that the content moves at the same speed as the finger and does not continue to move when the finger is released.

To avoid the confusion about what one-finger touching will do, an alternative gesture is available on modern devices—*two-finger scroll* and two-finger flicking, which work similarly to their one-finger equivalents. (Note that the one-finger version can work with stylus devices, but the two-finger gestures are always done with actual fingers.) Two fingers also provide the option of zooming (by pinching) or rotating the content, based on the movements of the fingers relative to each other, which is generally supported for contents that make sense—for example, maps can be scrolled, zoomed, or rotated using two fingers, but web pages can only be scrolled or zoomed, and contact lists can only be scrolled. As discussed in Section 2.5.1.11, using multiple fingers to manipulate graphical objects has a long history, including VideoPlace in 1985 [Krueger et al. 1985] and pinch-to-zoom in Rubine's work in 1991 [Rubine 1991a].

Touchpads in front of the keyboards on laptops (see Figure 3.4) now also support multiple fingers and these gestures for scrolling, zooming, and rotating. An important consideration for touchpads, which is also relevant to the scroll wheel on the mouse, is what content is scrolled? Scrollbars are attached to the window (or other content) that they will scroll, and direct touch on the screen makes it clear that the content under the touch is scrolled, but these do not apply to the scroll wheel or touchpad scrolling. There are two choices—(1) whatever window has the keyboard focus (typically the top window, where the user's typing will go; see a full discussion in Section 11.4) or (2) the content at the position of the pointer cursor, thus allowing users to scroll content that is *not* at the front. Modern systems all use option 2, with the scrolling following the pointer cursor, which follows the design first provided for the original scroll wheel on the Microsoft IntelliMouse from 1996 (Section 5.3).

Modern systems use the same feedback for these kinds of scrolling as described for flick-to-scroll, as shown in Figure 5.21.

5.10 Sliders

An on-screen *slider* is a widget for entering a single number in a range, by analogy with physical sliders discussed in Section 5.4. For example, the user might be

asked to enter a percentage between 0 and 100 for the opacity. Sometimes a slider widget looks just like a scrollbar, with increment buttons at the ends and a thumb for the current value, and other times it may have a different presentation with different features. A slider is sometimes called a one-dimensional input. Wikipedia says a synonym for a slider is a *track bar*, but I have never heard that term used. A percent-done progress bar can be considered a type of output-only slider, and these are covered separately in Chapter 6.

Typically, a slider will be parameterized in the same way as scrollbars (see Section 5.6), but they often add the minimum values since sometimes sliders will not start at 0 and may swap which end is the minimum value (see Figure 5.22).

One variant on sliders is to have *two* indicators on the same slider. This is useful when the user can specify a *range* of desired values, such as the times that flights should be between (Figure 5.22(d)) or the desired price range for a purchase. These are often called *range sliders* and were apparently invented by Ben Shneiderman, Catherine Plaisant, and their students in the Human–Computer Interaction Lab (HCIL) at the University of Maryland. Range sliders were described as future work in 1992 [Williamson and Shneiderman 1992] and first appeared in 1993 [Plaisant 1993].

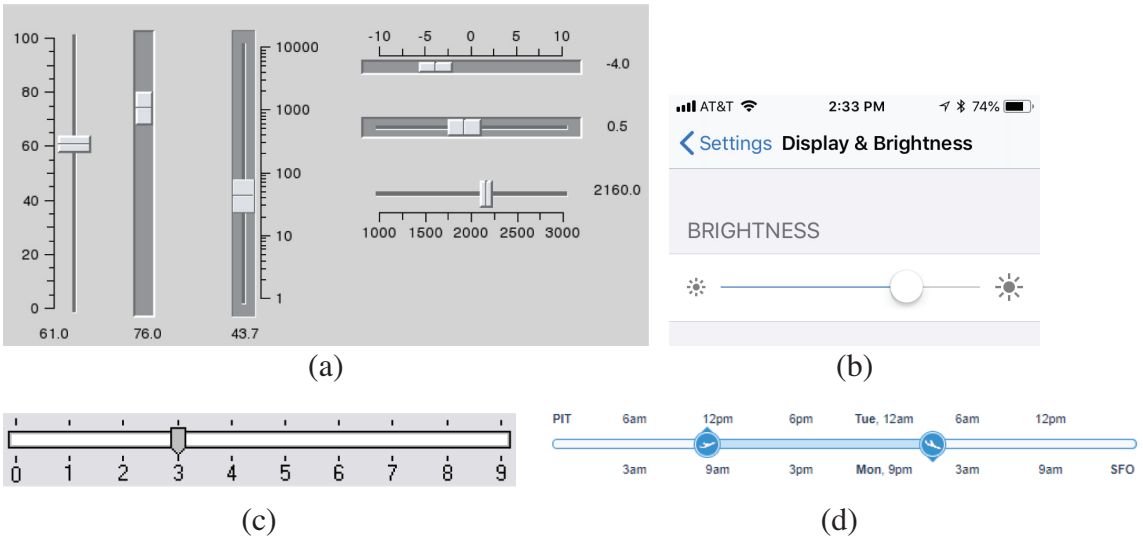


Figure 5.22 (a) and (c) Various sliders. (b) Slider on iPhone v11.3.1. (d) A current two-indicator range slider specifying the desired time range. Sources: (a) Qwt User's Guide 6.1.3, http://qwt.sourceforge.net/class_qwt_slider.html. (b) Screen capture from May 2018. (c) By Jaksmata—Own work, Public Domain, https://commons.wikimedia.org/w/index.php?curid=4587855. (d) Screen capture of www.hipmunk.com from May 2018.

A common use for sliders today is the play position for video players, which typically allows the user to see the percentage of the way through the video and to directly jump to any location. Modern video sites, such as Vimeo and YouTube, show a thumbnail (small picture) of the video as the user's pointer cursor moves back and forth along the slider (see Figure 5.23).

One variation on this, which was demonstrated in the 1994 *AlphaSlider* research system [Ahlberg and Shneiderman 1994] and has been used by the Apple iPad video viewer, allows the user to have more fine-grained control by pressing down on the indicator and moving vertically *away* from the indicator. Then movements left-and-right move the indicator proportionally less far. This is as if the finger or pointer was connected to the indicator with a bendable piece of metal. For very long content, this can make it much easier to move short distances.

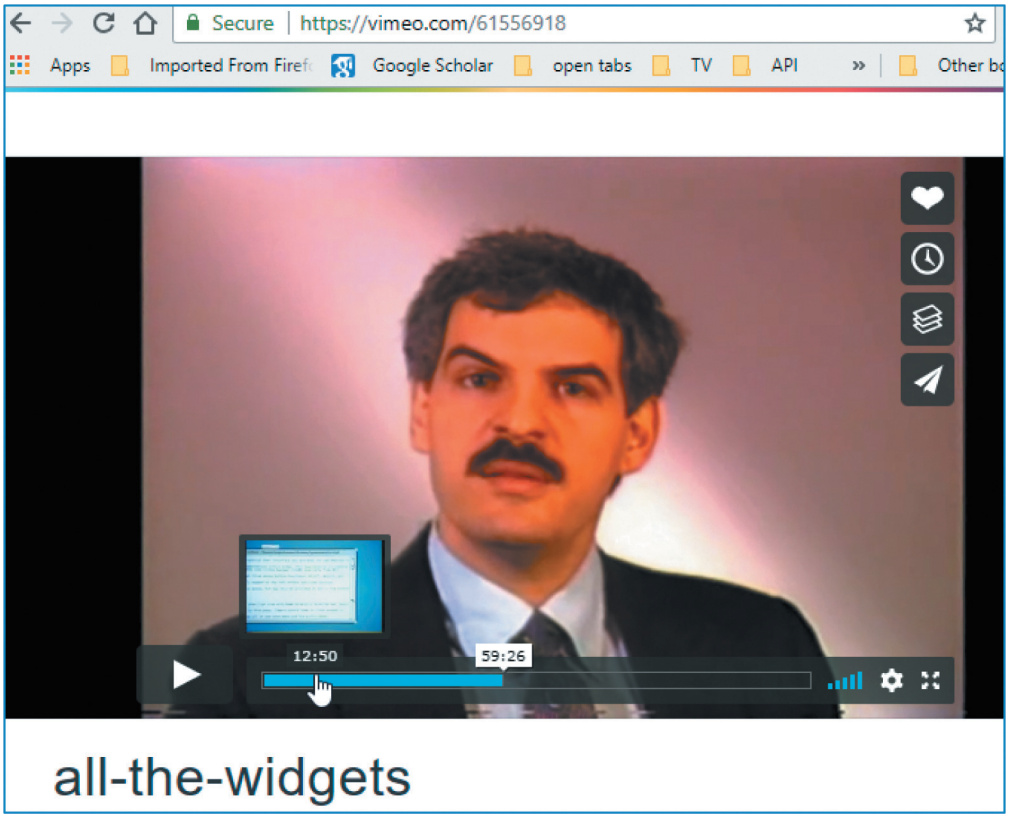


Figure 5.23 Video playback indicator at the bottom shows the current position (59:26) and a thumbnail of the video at the pointer position (12:50), and it can be used to monitor and adjust where it is playing. Source: Screen capture of Myers [1990b].

5.11 Research on Scrolling and Sliders

5.11.1 Evaluating Scrolling Speed

Given the importance of scrolling and sliders for computer use, it is not surprising that there has been academic research investigating them. For example, Bill Buxton and I did a study in 1986 on whether people would be able to scroll effectively to known locations with their non-dominant hand (e.g., their left hand) while using a mouse in their dominant hand to click on the targets once they were on-screen (see Figure 5.24). Given the timeframe, we had to create our own scrolling device hardware, and we discovered that people *did* use both hands for scrolling and pointing tasks and were faster than using conventional scrollbars [Buxton and Myers 1986]. I repeated this study in 2000 where we used scrolling widgets on the touchscreen of a PDA (similar to the ones shown in Figure 5.4) in the user's left hand, as shown in Figure 5.25, but we didn't see a speed increase in the two-handed version [Myers et al. 2000b]. Another study compared scrolling web pages for targets at unknown locations using four devices: regular scrollbars using the mouse, the regular mouse wheel, using the TrackPoint in the keyboard (Figure 4.10) with the left hand while using the mouse with the right hand, and using a mouse with a TrackPoint instead of a scroll wheel [Zhai et al. 1997]. In this experiment, the two-handed condition

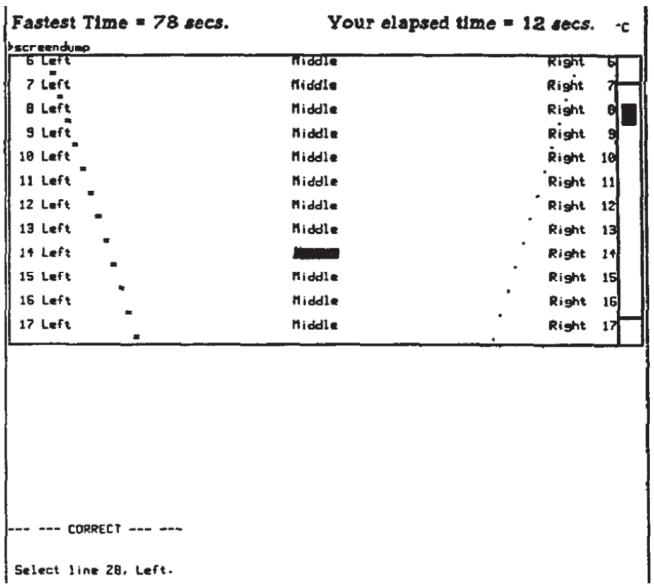


Figure 5.24 Tasks for scrolling to a known location. Prompt at the bottom told users how far they needed to scroll and which side of the screen it would be on. Source: Fig. 5 from Buxton and Myers [1986].

Figure 5.25 Using a PDA in the left hand to scroll while using the mouse in the right hand. Source: Fig. 1 from Myers et al. [2000b].

using the TrackPoint was 25.5% faster than the mouse, and using the scroll wheel was 8.7% slower.

Interestingly, there is no standard measurement method for scrolling, unlike for pointing tasks (Section 4.13). Therefore, the above studies had to invent their own materials. Some of the parameters that are important to capture in the tasks include:

- Scrolling where the user *knows the target location* versus when it is *unknown* so the user must visually browse while scrolling to find it [Cockburn and Gutwin 2009]. One would hypothesize that if the user does not know where the target is, slower scrolling would be used.
- Scrolling to locations that are *close or far away* when the user knows the target location. Different scrolling methods would be used to move a little versus a lot.
- Scrolling *interleaved with clicking*. If scrolling uses a scrollbar, then moving the pointer back and forth from the content to the scrollbar would be time-consuming. We have also found that since tapping on a smartphone is error-prone, the time to tap on an item dominates the time to scroll to that item, confounding the measurements of the scrolling itself. Whereas it is common that the user would need to interact with items after scrolling, in other cases such as reading or looking something up, just getting to the desired location is sufficient, and no clicking is involved.

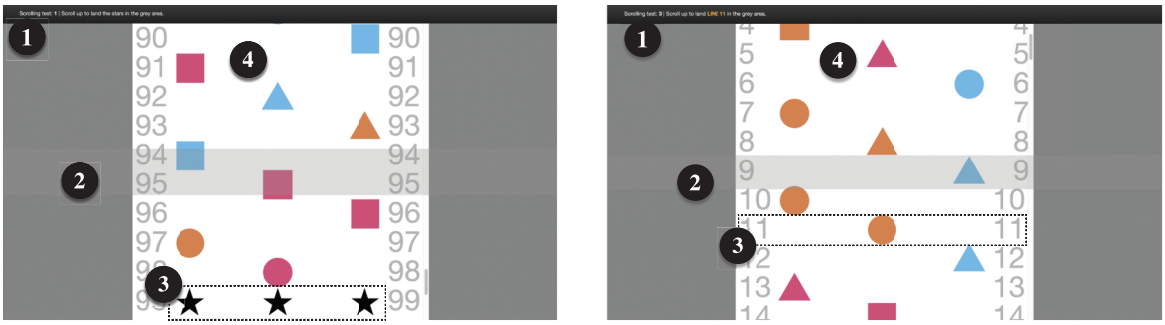
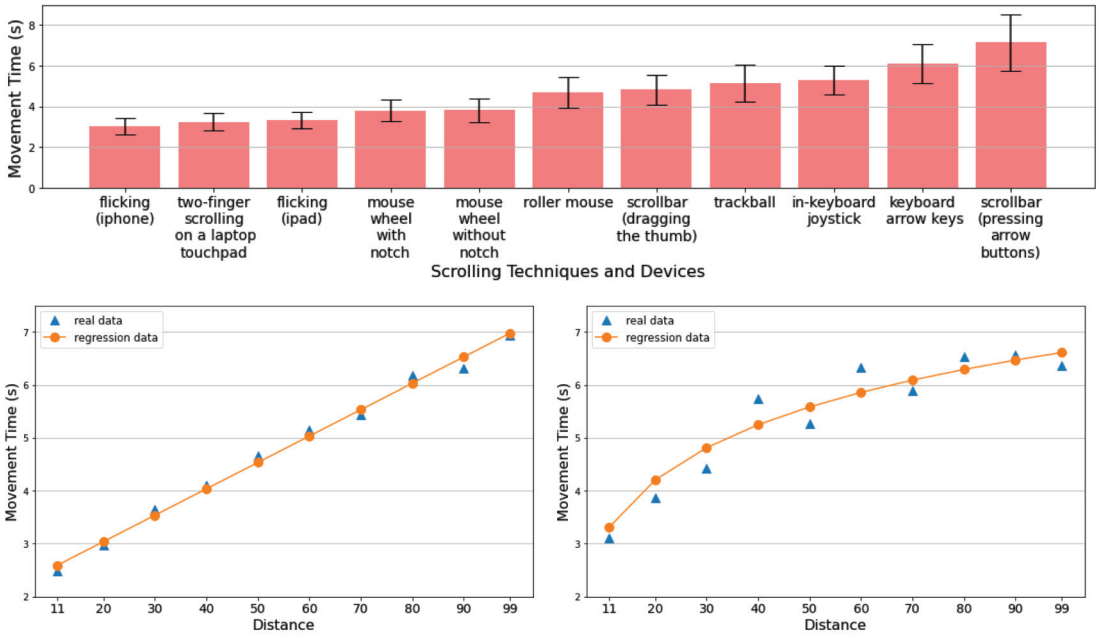


Figure 5.26 The user interface of ScrollTest contains two main pages: (a) Scrolling tests with *unknown* position target and (b) scrolling tests with *known* position target. Both pages have (1) instruction panel, (2) frame where the target line should be scrolled to, (3) target line, and (4) scrollable area. Source: Fig. 1 from Chen et al. [2023].

Based on these considerations, we have created a new scrolling test software called ScrollTest,[1] which can be used to evaluate scrolling speed (with or without the need for a click/tap). It is available from this book's website: www.ixtbook.com/scrolling/. For the non-click version, the user must scroll the target line into the gray area in the middle of the screen (see Figure 5.26). For the unknown case, where the user doesn't know where it is, the target line is a row of stars, so it is easy to see. For the condition where the user is told where it is, the target line number is displayed at the top of the screen. We tested ScrollTest across four dimensions: 11 different scrolling techniques/devices, 5 frame heights, 13 scrolling distances, and 2 scrolling conditions (i.e., with or without knowing the target position). The results show that flicking and two-finger scrolling are the fastest (see Figure 5.27, top)—flicking is also relatively precise for scrolling to targets already onscreen, but pressing arrow buttons on the scrollbar is the most accurate for scrolling to nearby targets. Mathematical models of scrolling are highly linear when the target position is unknown but like Fitts' law (Section 4.14.1) when known (see Figure 5.27, bottom). We also found that counting the number of *switchbacks*, where the user overshoots the target and switches direction (possibly multiple times), is a good measure of the *accuracy* of the scrolling technique [Chen et al. 2023].

1. The first design for ScrollTest was produced in my class as a final project by Cem Ergin, Emily Porat, Sijia Li, and Chun Wang in 2019. Then, Chaoran Chen as a TA in 2022 developed a new version that was formally tested and is described here. Some of these paragraphs and figures are excerpted from our paper [Chen et al. 2023].



(a) Regression analysis for unknown condition

(b) Regression analysis for known condition

Figure 5.27 (Top) Average movement time across all conditions (shorter is better) for all 11 devices tested using ScrollTest. (Bottom) Regression analysis of average movement time and distance across all devices—showing that the time was mostly linear when the target position was unknown (a), but matched a log scale like Fitts law when the target position was known (b). Source: Excerpts of Figs. 6 and 8 from Chen et al. [2023].

5.11.2 Research on Scrolling and Sliders

In addition to the influential research systems that were mentioned above, a number of other approaches have been studied to help make scrolling more effective. A persistent idea that has been tried multiple times both in research and commercially is *tilt-to-scroll*, where tilting a mobile device [Rekimoto 1996, Hinckley et al. 2000] or a watch [Rahman et al. 2009, Xiao et al. 2014, Shima et al. 2016] causes the content to scroll. Typically, the content seems to run downhill, so it would scroll down when tilted away from the user and up when tilted toward the user. Although this seems like an obvious idea, none of these systems have been successful due to three key factors: (1) it is hard to see the screen if it is tilted away from the user in either direction, so the user cannot see when to stop; (2) it is hard to control the speed of scrolling accurately, so users often overshoot, making this a fairly slow and error-prone technique; and (3) there can be many false-positives, where the device starts scrolling when the user does not want it to—for example, because the

user tilts the screen just to reduce glare or to move to a more comfortable viewing position.

A similar idea is to scroll when the user moves the device around in space, as if it is a viewer onto a large canvas that surrounds the user, where the content is fixed in space. An early system used a 3D tracker on the device [Fitzmaurice 1993], whereas more recent systems use the camera on the back of a smartphone to measure how fast and in what direction the camera sees the world moving [Wang et al. 2006]. Zooming can also be supported (e.g., for viewing a map) by moving the device toward and away from the user [Fitzmaurice 1993]. Another idea is to use the camera facing the user and use *gaze tracking* to automatically scroll whenever the user looks at the edge of the content [Sharmin et al. 2013], but this suffers from the *Midas problem* where the user's ordinary activities accidentally cause an unwanted action (here, trying to read the content causes scrolling), by analogy to the story of King Midas who accidentally turned his daughter into gold by touching her.

A totally different issue is helping users return to their previous place in the contents. As discussed in Section 15.6, the system's *undo* mechanism rarely works to undo scrolling actions and only undoes content edits. Therefore, users must manually try to get the contents back to their previous view—for example, if the user wants to "peek" at something elsewhere in the document, and then return—which some research has shown to often be problematic [Ko et al. 2006]. The *dwell and spring* interaction provides a clever way to enable scrollbars to return to the previous location by having a spring widget pop up if the user dwells after scrolling, and if the user moves the pointer onto the end of the spring, it *springs* the scrollbar back to the old position [Appert et al. 2012].

5.12 Summary of the Considerations for Scrolling

The sections above have discussed many different considerations that a designer must keep in mind if creating a new way to scroll contents. For convenience, they are summarized here:

- Scrolling a long way away (e.g., for a long document).
- Scrolling a short way (smallest increment—line, pixel, etc.).
- Scrolling to the ends of the contents, or a way to jump there.
- Feedback for where you are (what percentage of the way through).
- Feedback for how much you are seeing (fixed size or proportional-sized thumb).
- Scrolling up and down, versus left and right, versus supporting arbitrary directions such as diagonal.

- Which direction is "up"—consider the direction of the scrollbar arrows and the direction of moving two fingers on a touchpad.
- Scrolling with one finger, two fingers, or the non-dominant hand.
- Supporting auto-repeat and auto-scroll.
- Supporting continuous dragging of contents, which causes scrolling at the edges of the content.
- Is the scrolling abortable (e.g., by moving the pointer cursor outside the trough or hitting Esc)?
- Is the scrolling reversible and/or undoable?
- How fast can scrolling be done (e.g., based on a scrolling speed test)?
- Where to place the scrollbar? For example, which side of the window (left/right), but horizontal scrollbars are always on the bottom (and never on the top).
- Support for keyboard/physical buttons for scrolling (in addition or instead of using the pointing device).
- Support for direct and indirect pointing devices (some mechanisms do not support direct touch, including the OpenLook scrollbar).
- How do different scrolling mechanisms work together—for example, arrow buttons and scrollbars and swiping?
- How does the user control which content is scrolled (e.g., for scroll wheel or two fingers on touchpad)?

Consider also the different kinds of scrolling. Which of these will be supported by the system for scrolling?

- Up or down (or left and right), by one *increment*—usually one line.
    - Does this *auto-repeat?*
- Up or down (or left and right), by one *page* or one window-full.
    - Does this *auto-repeat?*
- Jump to the *start* or *end* of the content.
- To a particular *part* of the document by *percentage* (like dragging the thumb to be 1/3 of the way through the document).
- Up or down (or left and right) at a particular *speed* (rate-controlled scrolling).

- Directly *dragging* the content (as with the hand interaction or one- or two-finger scroll).
    - Can it be flicked and continue moving?
- Scroll so a particular *item* is in view (e.g., for auto-scroll or after a search).

5.13 Chapter Summary

This chapter has discussed the many ways to move around the contents shown on a device's screen, which is called *scrolling*. Many different interaction techniques have been developed for this, with the most popular being scrollbar widgets, a scroll wheel on a mouse, using one or two fingers to scroll, and flick gestures on touchscreens or touchpads. However, all of these techniques have advantages and disadvantages, so new techniques are still needed, especially when the size of the content is large. Scrolling in 3D (often called *panning*) is covered in Chapter 12.

CHAPTER 6

Percent-done Progress Indicators

6.1 Introduction

User interface textbooks (e.g., Shneiderman [1987] and Nielsen [1993]) and human interface guidelines (e.g., Apple Computer [1992, pp. 270–71] and Sherwin [2014]) have always recommended that users be kept informed when there will be a long wait. Nielsen recommends providing feedback if the wait is longer than 1.0s, and "for longer delays, users ... should be given feedback indicating when the computer expects to be done" [Nielsen 1993, p. 135].

A *percent-done progress indicator*, also called a *progress bar* or *progress indicator*, is a graphical technique that allows the user to monitor the progress during the processing of a task [Myers 1985b]. Progress indicators can be displayed on almost all types of output devices and can be used with many different kinds of programs. Practical experience and formal experiments show that progress indicators are an important and useful user interface tool and that they enhance the attractiveness and effectiveness of programs that incorporate them [Sherwin 2014]. Progress bars can serve as a key part of Nielsen's strategy to provide "visibility of system status" [Harley 2018]. HTML since v5 has a built-in `<progress></progress>` element.

Technically, based on my definition (Section 1.3.7), progress indicators are actually *not* interaction techniques as they aren't interactive: the user just watches them and doesn't "do something that causes a product to respond" (Section 1.3.7). However, progress indicators are included in most widget sets, and they are generally reusable across various applications. Further, I personally did an early (1985) study of progress indicators that showed that they were appreciated even when not entirely accurate [Myers 1985a, 1985b], which was reported to have been highly influential in their widespread use.

6.2 History

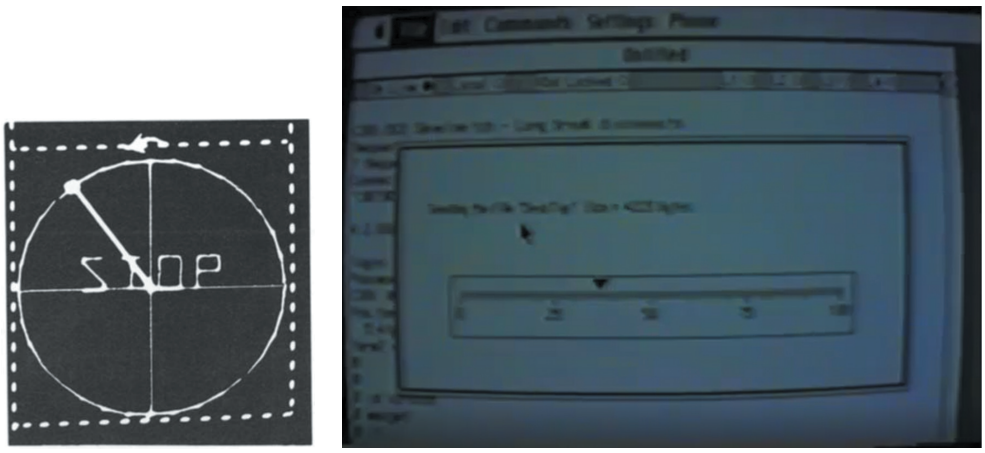
People have always wanted to know whether their machines are still working and how long it will be until they are finished, hence the flashing lights and reports about people listening for disk noises on classic computers. Other early examples to show that the computer had not crashed include having the program periodically rotate the print head of the teletype machines (Figure 2.3) to make a noise, or printing a dot or asterisk on a display screen or printer as the program made progress.

As soon as computers had pointer cursors whose picture could change, this was used to show that the computer was busy, and these are generally called *busy cursors*. Early examples of static pictures include a Buddha cursor from the University of Toronto to signify that patience is necessary (1975) [Tilbrook 1976, Baecker et al. 1979], an hourglass used by the Xerox Alto and Star (1970s and 1980s), and the watch from the original Macintosh (1984) (Figure 6.1(a)–(c)).

Somewhat more useful are *dynamic* pictures that at least show the computer has not crashed while it is busy, such as the flipping boxes from the Alto FTP program (1970s), which alternated between the two pictures in Figure 6.1(d) [Myers 1985a]. There was also a later version of the Macintosh watch (Figure 6.1(b)) where the hands moved around to show the computer was working. These have been called *random* or indeterminate progress indicators [Myers 1985b].

The first graphical presentation of an actual *percent-done* progress indicator that I know about was from 1976, and it described "a 'count-down' light-clock which ... tells the designer (a) that something is happening, and (b) how long it will take to happen, thereby removing two potential sources of panic" in a CAD–CAM application (see Figure 6.2(a)) [Spence 1976]. The original Macintosh used a progress

Figure 6.1 Pictures in the pointer cursor to show the computer is busy. (a) Static hourglass from the Xerox Alto and Star. (b) A Buddha cursor from the University of Toronto to signify patience is necessary. (c) The watch from the original Macintosh (1984). (d) Dynamic cursor from Alto FTP program. Sources: (a), (b), and (d) From Myers [1985a]. (c) Screenshot.



(a) (b)

Figure 6.2 (a) A count-down light clock from 1976. (b) The original Macintosh MacTerminal progress bar. Sources: (a) Fig. 4 from Spence [1976]. (b) Myers [1985a].

bar for file transfers in the MacTerminal program (1984; see Figure 6.2(b)) [Myers 1985a].

When I designed user interfaces for the PERQ computer (Figure 2.5) starting in 1980, I designed in the ubiquitous use of progress indicators, both for random progress and percent-done progress indicators [Myers 1985a]. When I designed the Sapphire Window manager (Figure 11.1) in 1983, I built in a progress bar as part of the window title and also one or two progress bars in the icons for each window [Myers 1984]. The progress bars could show random (indeterminate) progress (Figure 6.3(a)) with a flickering pattern. For percent-done progress bars, there were two bars to support hierarchical tasks where there might be subtasks with their own progress, in addition to the overall task percent complete (Figure 6.3(b) and (c)).

(a) (b) (c)

Figure 6.3 Three icons from the Sapphire window manager. (a) Random progress. (b) and (c) Regular progress bars with two or one level. Source: Myers [1984].

However, progress indicators did not seem to be widely used, so we decided to do a study to try to foster more widespread use. In particular, we wondered if people would prefer to have progress indicators, especially when the task performance time varied significantly. Not surprisingly, people did prefer to have the progress bar in all situations, but we found no significant distinction when the task time varied [Myers 1985b]. This study was influential at the time and was credited with motivating the use in more situations, even when the system could not accurately predict the actual percentage of the way through the task [Engber 2014].

6.3 Implementation

The original progress indicators required that the programmer insert calls to update the widget inside the inner loops of the code. For example, the code would recalculate the percent done when each packet came in for a file transfer over the network and update the widget. The code would also have to know the total size first, so this motivated protocols that would first identify the total length in the first header (e.g., for a file retrieval). When the total size was not known, the code would at least update the random progress in the inner loop. This was especially important for single-process machines.

Now, at least for random progress, the indicator can update itself. Typically, the indicator is an animated gif or other tiny movie, and the operating system causes it to animate. This somewhat defeats the purpose in that it does not show when and if the application process is actually making progress, but it at least shows that the computer and operating system are still running.

6.4 Variations

There have been many variations in the way the progress indicators have been portrayed, and the extra information around them. Naturally, users prefer when they also have specific time estimates displayed (e.g., "3 minutes remaining") alongside the progress bar (Figure 6.4). When computers could handle more sophisticated graphics, animations were often added to the indicators. For example, starting with Windows 95, Microsoft had an animation of a file flying from one folder to another along with the progress bar for copying and transferring files (Figure 6.4). The bar itself has often been animated—for example, with a shimmer or glow effect. Research showed that careful manipulation of this animation could cause the progress bars to *appear* faster so people felt like it was taking less time [Harrison et al. 2007, 2010a].

For random progress, the bar might be replaced with a "barber pole" that spins in place, or the indicator may bounce back and forth instead of growing from one

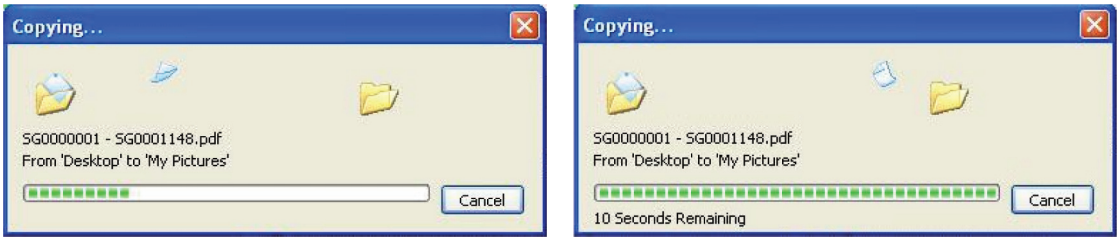


Figure 6.4 Two screenshots of the progress dialog in Windows XP. The blue and white paper repeatedly flies from the left folder to the right one. This design dates back to at least Windows 95 (1995). Source: Screen captures from Windows XP SP2, 2002, used with permission by Microsoft.

end. Busy cursors also evolved to have more animation. For example, the Macintosh changed the busy cursor to be a spinning color wheel in 1999, sometimes called the *spinning beach ball* or *spinning pinwheel* (officially, the *Spinning Wait Cursor*). Previous and later versions have had different animations. Most systems now seem to use variations on rotating parts of circles for their random progress indicators.

Microsoft Windows now provides scrollbars by coloring the background of taskbar entries (see Section 11.7), so the user can monitor the progress of activities even for windows that are not visible (Figure 6.5(c)), like Sapphire did back in 1983. When searching or doing other long tasks in the Windows File Explorer, the folder field acts as a progress bar (Figure 6.5(b)) so extra space is not needed. These both started with Windows 7 in 2009 (or possibly earlier). The Windows 10 progress bar provides a speed gauge (Figure 6.5(a)).

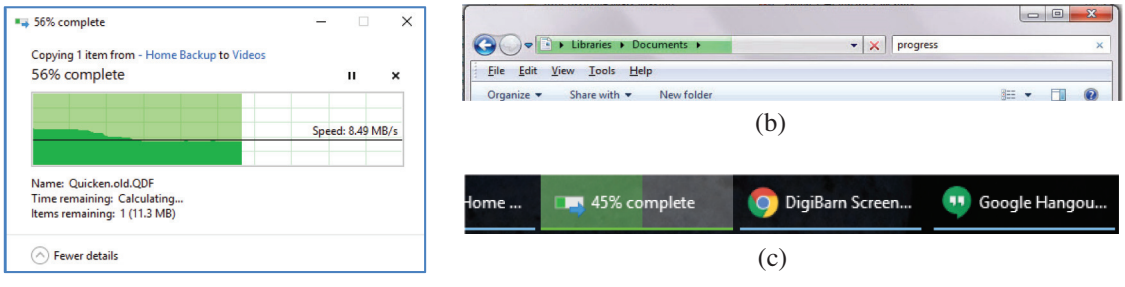


Figure 6.5 (a) Windows 10 progress dialog. (b) Windows 7 progress in folder field—the green shading fills from the left to the right end of the field. (c) Windows 10 taskbar progress. Source: Screen captures of (a) Windows 10, (b) Windows 7, (c) Windows 10 in May, 2018. Used with permission by Microsoft.

6.5 Design Considerations and Conclusions

Although computerized devices continue to get faster, there will always be tasks that take a noticeable amount of time. In these cases, it is always important for the device to show that it is still processing and has not crashed and to show the user how long it is expected to take. Previous time-consuming tasks, like file transfers, are often now instantaneous, and for tasks that are slower tasks now (e.g., artificial intelligence and network-based tasks), it may be harder for the programmer to accurately predict the completion time. However, good usability still requires the systems to provide as accurate estimates of the finishing time as possible [Myers 1985b, Sherwin 2014]. It will also be important to adapt progress indicators for devices without displays, such as *smart speakers* (also called *voice assistants* for the home, see Section 18.2), since they may eventually be able to take on long-running tasks.

CHAPTER 7

Menus

7.1 Definitions

Menus are among the most important and varied kinds of interaction techniques. I have defined a menu as a widget that allows the user to choose one or more commands or values from a set of options [Myers 1990b]. Wikipedia's (en.wikipedia.org/wiki/Menu_(computing)) current definition (as of 2023) is also quite good: "In user interface design, a menu is a list of options presented to the user." This covers many different kinds of widgets—including menus, buttons, palettes, radio buttons, checkboxes, option buttons, and so on—these will all be defined and explained in this chapter.

Menus do *not* include the following: selecting objects in a graphics editor (e.g., so they have handles that can be used to modify their shape), selecting cells of a spreadsheet (see Chapter 10), selecting or entering text (typing or editing text, see Chapter 8), or selecting a value through the use of a slider even if the slider provides a small number of choices. A *tree view* (Section 10.11)—for example, as used in a file browser such as the Windows "File Explorer" or the Macintosh "Finder"—is also not a menu. Sometimes menus and other widgets are grouped into *dialog boxes* or *forms* (discussed in Chapter 9), which are not menus themselves even if the contents are only menus. Similarly, I do *not* count "picker" widgets as menus, like a color picker or date picker (discussed in Chapter 9), since pickers usually include a variety of widgets, only some of which are menus (e.g., sliders for color values or text entry fields for the time).

Menus have been a key interaction technique since the beginning of computing. Choosing one or more from a set of options is one of the fundamental interaction operations [Foley and Wallace 1974, Myers 1989] and appears prominently in theories of interaction (Chapter 19). The main advantage of menus, and one of their defining characteristics, is that the list of options is *presented* to the user. This is typically *displayed* on a screen, but menus can also be auditory, where the options

are spoken ("press 1 for sales, 2 for service, ..."). This takes advantage of "recognition rather than recall"—as Nielsen says: "Minimize the user's memory load by making objects, actions, and options visible" [Nielsen and Molich 1990].

Menus come in hundreds of variations, with a corresponding number of design choices that designers can make—remember all the aspects of the checkbox example in Section 1.1, and that didn't even consider using different widgets instead of a checkbox. My "All the Widgets" video from 1990 devotes about 44min (about 1/3 of the whole running time) to various kinds of menus, and many new variations have been used since then. Consequently, this chapter cannot cover *all* possible variations and considerations, but hopefully it will capture the main dimensions.

7.2 Different Categories of Menus

Depending on how the menus are organized and presented, they have different names. This section summarizes and defines the key kinds. Note that the full descriptions of the variations for these kinds of menus are discussed later in this chapter.

- *Menus*—although this chapter is using the word "menu" for the general category of widgets supporting choosing among options, some contexts use the word "menu" in a more restrictive manner—to only refer to widgets that display a list of options where the options are visible and the full option itself is highlighted to show it is selected (see Figures 7.1, 5.11, etc.).
- *Submenu*—a submenu is any menu that appears as a result of a previous menu item being selected (Figure 7.4).
- *Popup Menu*—a popup menu (Figure 2.10(b)) is a menu that starts out not visible and becomes visible as the result of a user action, typically the user pressing a mouse button or on a touchscreen. After the user is finished using the popup menu, it disappears again.

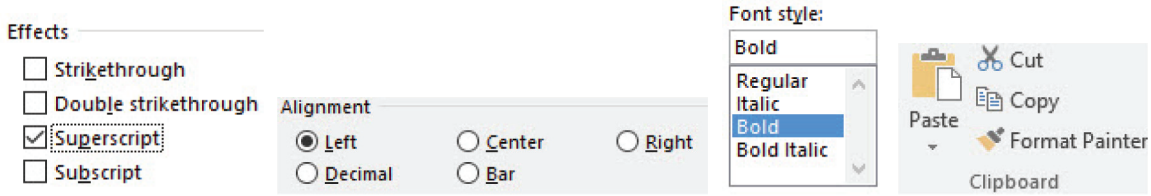


Figure 7.1 (a) Checkboxes, (b) radio buttons, (c) font style submenu popped up from a combo box, and (d) clipboard menu in Microsoft Word 2016 (part of the Office Ribbon). Source: Screen captures, May 2018. Used with permission by Microsoft.

- *Fixed Menu*—a fixed menu (Figure 7.3) is generally visible (as opposed to a popup menu). Fixed menus may actually come and go based on context or user actions, but when the user intends to operate them, they are already visible on the screen.
- *Context Menu*—a context menu is a particular kind of popup menu that appears when the user clicks on an object or the background with a special signal (e.g., the right mouse button or a *long-press* on a touchscreen). Typically, the items in the menu depend on the context (hence the name) including the location of the click (what is under the pointer) and global state (such as if there is a selected object).
- *Hierarchical Menu*—a hierarchical menu is the general name for widgets where choosing one menu item may cause a new submenu to appear. Technically, a "hierarchy" should be a *tree* of menus, but some hierarchical menu structures are actually a *graph* in that some items may appear in multiple menus, and thus there may be different paths through the menus to the same item.
- *Cascading Menu*—a cascading menu (Figure 7.4) is a particular kind of popup hierarchical menu where the submenus pop up beside the parent menu, possibly for multiple levels.
- *Menu Bar* or *Drop-Down Menu*—a menu bar (sometimes spelled *menubar*; also called a *drop-down menu*; see Figure 5.11 and many others) is a specific kind of hierarchical menu where the first level menus are in an always-visible bar (typically horizontal), and the first level submenus popup below the selected first level item. The menu bars on the Apple Lisa (Figure 5.11) and Macintosh and Palm (Figure 7.7(a)) are at the top of the screen and in Microsoft Windows (Figure 2.17) are at the top of each window. The top-level of the Microsoft Office *Ribbon* (Section 9.5 and Figure 7.1(d)) counts as a menu bar, even though the ribbons themselves are not menus—being more like dialog boxes since they collect together a variety of widgets.
- *Combo Box*—a combo box (also called a *drop-down list* or sometimes an *option menu*) is a widget that always displays the current value, and when the user clicks on the widget, a popup menu shows the list of options (Figure 7.2(b)–(d)). (The original Macintosh calls this kind of widget a "PopUp Menu" [Apple Computer 1992, p. 82], but this name is too ambiguous.) When a new option is selected, the menu disappears, and the widget displays the new value. Sometimes the original widget is a text entry field, so users can either type the new value or select it from the submenu. Thus, this

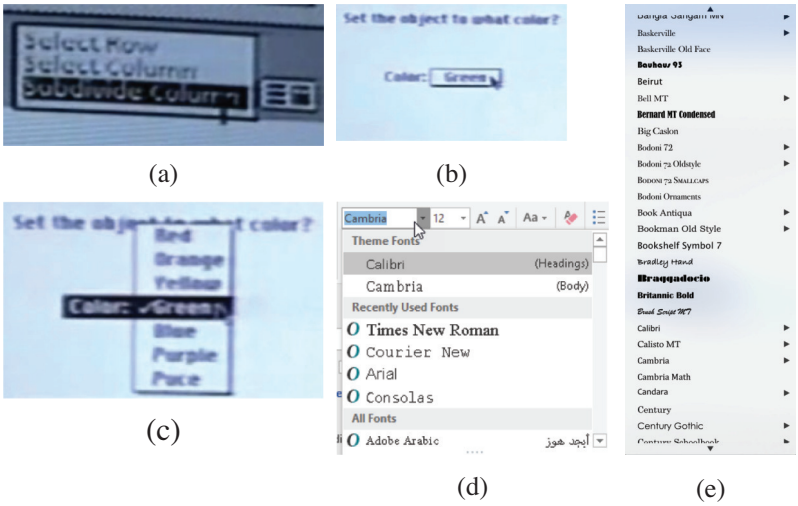


Figure 7.2 (a) Xerox Star menu button and its menu. (b and c) Original Macintosh combo box before and after pressing on the button. (d) A combo box supporting text entry, a menu with a scroll bar, and a "Recently Used Fonts" section. (e) Menu from Microsoft Word for Mac with scroll arrows at top and bottom. Sources: (a) Myers [1990b] around timecode 21:35. (b and c) Myers [1990b] around timecode 28:37. (d) Screen capture of Microsoft Word, June 2018. (e) Screen capture of Microsoft Word for Mac, May 2019. Used with permission by Microsoft.

widget is a *combination* (hence "combo") of a button or text field and a menu. Sometimes, the submenu "drops down" from the widget but other times the submenu is centered on the widget, so "drop-down" may be somewhat misleading. However, some systems (e.g., Java and .NET) distinguish a combo box from a drop-down list by limiting the former to when the widget is a text entry field and the latter to when the original widget is just a button. This widget is different from a *menu button* (see below) in that this one displays the current value whereas the menu button is a static icon or label.

- *Palette*—a palette (Figure 5.11) is a fixed menu, usually of icons (e.g., the drawing mode) or properties (like colors or fonts), that controls persistent modes or options. The name "palette" is a generalization of color palettes. Palette widgets typically do *not* contain commands (like "delete") but do contain mode settings (like picking the "eraser" tool so the user can start deleting).

- *Tear-Off Menu*—a tear-off menu (Figure 7.15(a) and (b)), also called a *pinned menu* (Figure 7.15(c)), is a submenu or popup menu but on which the user can perform some action to cause it to instead be a fixed menu that is always

visible. A different action can remove it from the screen so it must again be accessed as a popup. An example use is to convert a submenu of textures into a palette (Figure 7.15(b)).

- *Pie Menu* or *Marking Menu*—a pie menu (Section 7.3.9.1) is a menu where the items are arranged around a circle rather than as a row or column or grid. Typically, each item gets an equal-sized portion of the circle and looks like a pie slice. Pie menus are typically popups but can also be fixed. A *marking menu* is a variant where the user can move from menu to submenu without releasing the pointer button.
- *Accordion Menu*—an accordion menu is a (typically) two-level hierarchical menu where the top level is a fixed menu and selecting a top-level item causes the submenu items to slide out below the top-level menu item, pushing the other top-level items down. Typically, only one submenu is opened at a time, so selecting a different top-level item closes the previous submenu and opens the new one. Thus, the whole menu grows and shrinks like an accordion.
- *Spin Menu* (Figure 7.7(b))—a spin menu, sometimes called a *spinner* or a *roller menu*, uses the metaphor of a wheel with the options printed on it, which rotates (spins) when the user moves the pointer up or down, like the rollers on a mechanical slot machine. (Note that Wikipedia uses the word "spinner" for a kind of number entry widget, see Section 8.7.)
- *Audio Menu*—an audio menu or *phone tree* is a menu that is spoken aloud rather than appearing on a display, such as "press 1 for sales, 2 for service," and so forth. Typically, these are used by automated assistants reached through the phone, and they are usually hierarchical (after each selection is a further submenu), hence the name "phone trees."
- *Button*—a button (Figure 7.12, etc.) is a widget that triggers an event or designates a single value. Buttons can be part of a set, where each one corresponds to a different event or value. Buttons can be physical (such as on a machine) or virtual (such as graphical elements on a screen). A difference between graphical buttons and a regular menu is that buttons are graphically depicted as separate, whereas menu items are depicted together.
- *Key*—a key on a *keyboard* or *keypad* is a form of button, usually arranged as part of a set. A *keyboard* typically contains keys for all alphabetic characters, whereas a *keypad* would have a much more limited number of keys—for example, for only the numerals. (Text entry using keys is covered in Chapter 8.)

- *Menu Button*—a menu button refers to a button containing a picture, usually the *hamburger icon* (three horizontal lines), that when pressed pops up a menu (Figures 5.10(b) and 7.2(a)). This was first used on the Xerox Star but is now widely used by smartphone user interfaces. See Chapter 26 for a discussion about the origins of the hamburger icon.
- *Soft Key*—soft keys refer to on-screen labels that are displayed near physical buttons to say what they do. For example, there may be a row of labels at the bottom of a computer screen to show what the keyboard function keys do (see Figure 5.6(a)), or labels on an automated teller machine (ATM) for a bank might have labels on the screen for what the physical buttons around the screen do.
- *On-Screen Menu*—an on-screen menu refers to menus on consumer electronic devices where the menu is displayed on a screen but selection happens through a remote control. For example, a cable box would display menus on the TV screen that are navigated and selected using keys on the remote control (Figure 4.4(b)).
- *Checkbox*—a checkbox (Figures 1.1, 7.1(a), and 7.9) provides for an on–off (yes–no) choice, sometimes with additional options like *partially* on (see Section 7.3.1). When checkboxes are part of a set, all the options are independent, so users can choose zero, one, or more options. Typically, a label is located to the right or left of the graphic for the box. As discussed in Section 7.3.7, the mark that indicates whether the checkbox is on or not is sometimes not an actual checkmark but can be an "X" or other indicator. The term comes from paper forms where users would draw a checkmark in a square.
- *Toggle Switch*—a toggle switch (sometimes just *switch*) provides a small set of values (typically two: on–off or yes–no) and thus can be used like a checkbox. Typically, toggle switch widgets (on a screen) have a graphic presentation that looks like a physical slide switch (Figure 7.11) and will usually have a label to the left or right to explain what it is for (Figure 7.9(d)).
- *Radio Button*—a radio button (Figures 7.1(b) and 7.10) provides for exactly one selection from a set of options. (Rarely, radio buttons can support zero or one selection.) When a different radio button from the set is selected, the previous one that was selected becomes de-selected. Typically, radio buttons are represented on screens with a circle graphic that fills in when selected, with a label to the left or right to identify what it is for. The name comes from old automobile radio mechanical channel selectors, where one button would

pop out when another was selected. The name was invented by Ted Kaehler for these types of widgets in the Smalltalk system in 1974 or 1975.[1]

7.3 Dimensions

Although many of the differences among kinds of widgets were discussed above, it is important to more explicitly consider the various dimensions on which menus can vary. These dimensions will help designers determine which kind of menu is most appropriate given different situations and will affect other aspects of the design.

7.3.1 How Many Options Can Be Selected?

The first decision is whether the user needs to be able to select a single option or multiple options. For example, checkbox widgets (Figure 7.1(a)) allow zero, one, or more items to be selected; radio button widgets (Figure 7.1(b)) allow exactly one choice; and menu widgets (Figure 7.1(c)) can support either. For menus where the user issues a command, like for Cut or Copy (Figure 7.1(d)), only one option can be selected, but menus that list data often support multiple selections. Sometimes, checkboxes may need to represent *partially* selected—for example, when some of the text is strikethrough and some is not (Figure 1.2), and sometimes radio buttons need to have the ability to have no items selected (zero or one but not more selected). Other times, radio buttons and other menus start off with no selection, but once the user makes any selection, the user can change what is selected but cannot go back to there being no selection.

For menus that support multiple selections, often using the Shift key will select all the items in a range from the original selection through the item Shift-clicked on. Using Control-click on an item will usually toggle the item in the selection.

7.3.2 How Many and What Kind of Options Are There?

Another key decision when selecting a type of menu is how many options need to be presented. When there are under 6 to 10 options, it may be feasible to use radio buttons and checkboxes, but if there are 10 to 100, then a menu may be better, and if over 100, typically a text input field would be preferred [Apple Computer 1992]. Menus in dialog boxes would typically take the form of menu buttons or combo boxes (as in Figure 7.1(c)), rather than a fixed menu, to save space.

1. Personal email from Ted Kaehler to me on June 16, 2018: "Yes, I did coin the term Radio Buttons, after the station buttons on old car radios. The [software object-oriented] class was for a painting system where each tool was made up of one size brush, one BitBlt painting mode, and other choices. I clearly remember sitting in front of an Alto in Building 35 trying to think of a good name for the class. This was in Smalltalk ... in [1974 or] 1975 (I think)."

The next question is whether the options are a flat list (like choosing a font name) or grouped into categories (e.g., top level "File" and "Edit"). Often a hierarchical presentation, such as menus with submenus or a menu divided into sections with dividing lines, will be used when there are categories. In a hierarchical presentation, one design decision is whether the category name itself can be selected to cause an action (beyond causing the submenu to appear). For example, on many web pages, hovering over a category name pops up a submenu of items, but clicking on the category name goes to a webpage for that category.

If the items are in a flat list and there are too many to fit on the screen, then some kind of scrolling for the menu items must be supported. Options include providing a regular scrollbar in the menu (as in Figure 7.2(d)) or providing arrows at the top and bottom of the menu (Figure 7.2(e)).

7.3.3 Fixed or Popup?

There are many kinds of menus that are fixed on the screen so the user can always see the options (Figure 7.3) and other kinds where the menus popup (appear) when the user does some action, like click a particular mouse button on an item or the background. For example, the Smalltalk menu in Figure 2.10(b) appeared with a mouse button click on the text, and the menus in the Xerox Star appeared with a mouse button-down on the menu button (the "hamburger icon") shown in Figure 5.10(b).

Fixed menus take more space on the screen but are faster to use, and they keep all the options visible to users. Some menus are hybrids—for example, menu bars

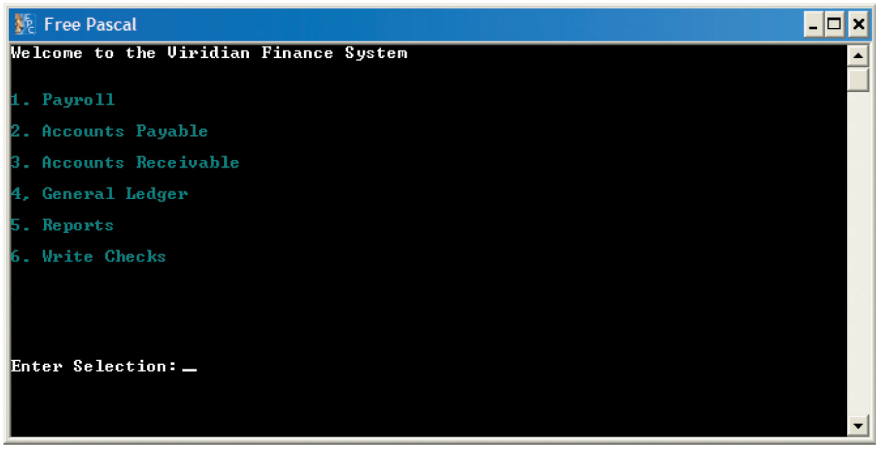


Figure 7.3 Fixed textual menu. User makes a selection by typing on the keyboard. Source: By Rfc1394—Own work, Public Domain, https://commons.wikimedia.org/w/index.php?curid=10479195.

(Figure 5.11), which first appeared on the Apple Lisa, have a fixed menu across the top of the screen, and each item in that menu pops up a submenu.

7.3.4 Invalid?

The Xerox Star (and most previous systems) *removed* items that were not relevant from the menus. For example, if nothing is selected in a graphics program, then a *delete* operation may not make sense. However, the Apple Lisa and Macintosh decided to leave all menu items in the menus all the time, and just gray them out to show they are currently invalid (see "Reshape" in the Edit menu in Figure 5.11). Other systems just leave all items available in the menus and provide an error if an invalid menu item is selected. There are tradeoffs with all of these designs.

Removing and graying out items make sure that the user can never execute an illegal operation, which helps enforce Jakob Nielsen's guideline of "*Error prevention*: Even better than good error messages is a careful design which prevents a problem from occurring in the first place" [Nielsen and Molich 1990]. The advantages of *showing* the items grayed out instead of eliminating them are that the user can more easily remember where menu items are if they are always viewable (providing consistency of location for menu items as well as keeping the menus consistent), and the user is cued that those operations exist but are not available. The advantage of *removing* invalid items is that this makes the menus smaller, and thus easier to navigate. Also, for some menus it is impractical to list all possible values. For example, the font name menu only shows the fonts currently installed and not all the possible fonts. Similarly, the font size menu does not show all possible sizes.

In most systems, the highlight that follows the mouse skips over the grayed out items (so they cannot be selected), but an augmentation that has been available in some systems is that the user *is* allowed to highlight grayed out items, and a message explaining why the item is illegal is displayed in a window status bar (typically at the bottom of the window, see Section 16.2) or in a *tooltip* that pops up (Section 16.3). Microsoft Windows now shows the tooltip for grayed out items when the cursor hovers over them, but uses the same message whether the item is grayed out or not, so it doesn't directly help understand why it is invalid.

The advantage of leaving all items selectable is that sometimes the run-time error messages can be helpful for people to understand why that operation is not available, possibly clearing up the user's misconceptions. This is especially useful when the grayed-out items cannot be highlighted. Also, sometimes it is infeasible to pre-calculate all the items that should be grayed out. For example, most web pages do not gray out product categories that are currently empty since it may be too slow to determine all of them before the user's selection.

7.3.5 Cancellable?

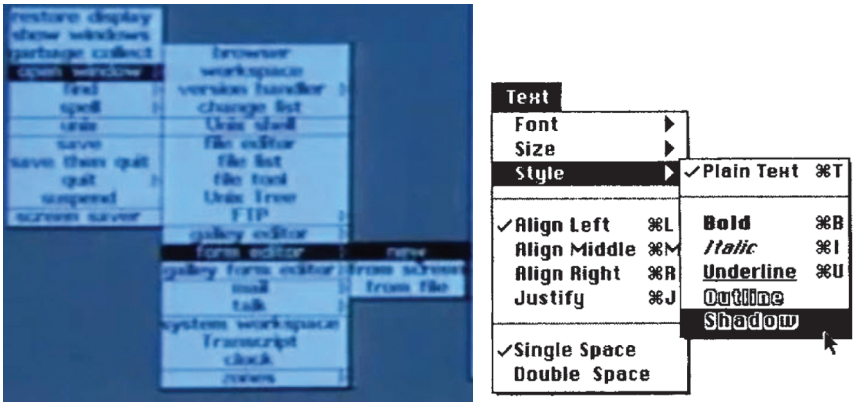
Virtually all menus allow the operation to be aborted (*canceled*) with no selection if the user moves the pointer away from all items before clicking or releasing. For example, pressing down on a menu item or a radio button and moving outside the menu before releasing will abort and cause nothing to happen. Similarly, if a popup menu is displayed, clicking or touching outside of the menu usually causes it to go away without executing any commands. Sometimes, there is an extra way to abort menus—for example, by hitting the Esc key on the keyboard.

For menus operated by the keyboard or through physical buttons, it may be harder to abort. Remote controls for consumer electronics sometimes have a "back," "return," or "exit" key that will abort the current menu without changing values (see the "EXIT" and "RETURN" buttons on the remote of Figure 4.4(b) and the semi-circle button on the bottom left of Figure 4.5 for going back in the Mazda menus). In other cases, the designer will need to include an explicit *back* option as one of the menu items. See also Section 15.4 about aborting in general.

7.3.6 Submenus

Whenever there are more than a few items, it is generally necessary to divide the menu items into a hierarchy of menus, where choosing an item from a menu will bring up a lower-level *submenu*. There are many different ways that submenus may appear. Early menus just replaced the higher-level menu with the submenu items [Newman and Sproull 1979], which has the problem that users can get confused about what level of menu they are at and how to abort or return to higher-level menus. Most later systems popped up submenus to the right of the parent menu (except when the right edge of the screen is reached, in which case they often go to the left), often in long chains for deep trees (Figure 7.4(a)), which have been called *cascading menus*. An advantage of displaying all of the parent and submenus is that users can more easily change which submenu is displayed. As mentioned above (Section 7.3.3), menu bars display the top-level menu as a fixed horizontal menu, with the first level submenus dropping down from there. There can be second and further level submenus that pop out to the right of the previous menu (Figure 7.4). Web pages use a variety of behaviors and displays for menus and submenus—for example, for categories and products in those categories, as chosen by the web page designer.

Section 7.3.11 discusses the overall behavior of menus (how they are operated with a mouse and keyboard), and Section 7.5 discusses how to organize the contents across the menus and submenus. In general, submenus are either invoked by pressing or moving to the parent item or else the user may be required to slide



(a) (b)

Figure 7.4 Examples of "cascading" submenus: (a) submenus in Smalltalk can be arbitrarily deep. However, (b) the original Macintosh submenus are recommended to be only one level deep. Sources: (a) Myers [1990b] around timecode 20:20. (b) Apple Computer [1992, p. 79].

to the right out of the parent item, which can require the user to move carefully to avoid sliding into the adjacent items by accident.

7.3.7 Graphic Design

Obviously, a key difference between the different kinds of menus is what the selected item looks like. When computer displays were all monochrome (black and white), it was typical for the feedback for menus to be *reverse video*, meaning that if the usual display was black text on a white background, then the selected item would be the reverse, or white text on a black background, as shown in Figure 5.11. A similar reversing strategy can be used in color or else the highlight can be sufficiently similar to the background color so that the item text color does not need to change (Figure 7.5). Alternatively, the foreground color of the selected item might be changed, as in the hand icon in Figure 5.19(a) in the menu at the top, which is drawn in blue to show it is selected. The highlighting for buttons is usually similar, as is the highlight for menus and buttons when the items are pictures or icons instead of text (e.g., see Figure 5.19(a) and the selected arrow in the palette menu in Figure 5.11). To show the selected item in a menu bar submenu, the Macintosh displays a checkmark (Figure 7.6).

Menus on portable devices have typically used the same design to highlight the selected item. For example, the menu bar on the Palm Pilot (Section 3.13) looked just like desktop menus (Figure 7.7(a)), and the Windows CE and PocketPC (Section 3.14) menus were similar. On the iPhone, menus are rare—there might be

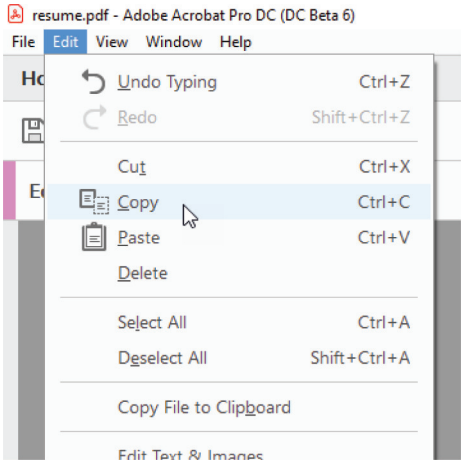


Figure 7.5 Menus in a recent version of Adobe® Acrobat® Pro DC (Beta 6). The top-level menu selection is shown with a blue background and the text changes color to be white, whereas submenus use a light blue background, and the text does not change color. Source: Adobe product screenshot reprinted with permission from Adobe.

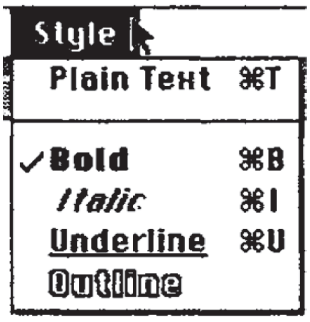


Figure 7.6 Original Macintosh menu bar with selected options in a submenu shown with a checkmark. Source: Apple Computer [1992, p. 65].

lists of options, each of which goes to a different page of options (as in Settings), or sometimes a spin menu will be used (Figure 7.7(b)).

Naturally, the indication of the selected item in a checkbox is usually a picture of a checkmark (see Figure 7.1(a)), but it has also been an "X" (e.g., in the original Macintosh), a square (see Figure 7.9), or just filling in the widget (e.g., in the Xerox Star, see Figure 7.8). Radio buttons have usually been represented as a circle, either filled in or not, but have also been a diamond (Figure 7.10). The original Xerox Star used boxes that were attached to represent options where only one could be selected (like "Show" in Figure 7.8), and detached boxes that could be turned

(a) (b)

Figure 7.7 (a) Menus on the Palm Pilot. (b) Spin menu on iPhone iOS v.3. Sources: (a) Screen capture of Palm Pilot. (b) O'Reilly Group, fair use, https://www.safaribooksonline.com/library/view/beginning-iphone-3/9781430224594/ch07.html.

on and off independently when multiple items could be selected, like "Justified" in Figure 7.8. The Star called these *choice parameters*.

For buttons, the active area where the user can click or tap is usually the boundary of the button. Menus should have the active area be the full width of the menu and the height of the item (some poorly designed menus only have the text extent be the active area). For checkboxes and radio buttons, it is preferable if the textual label is part of the active area that can be clicked and not just the icon representing whether it is on or not.

Researchers explored different presentations for toggle switches for touchscreens back in 1990–1992 (Figure 7.11) and showed that people could figure out how to operate them without instructions [Plaisant and Wallace 1992]. Most of the settings on an iPhone are controlled by toggle switches (Figure 7.9(d)) that originally users needed to swipe across but later changed so you only need to tap on them. These toggle switches essentially operate like checkboxes (each is independent). In a few cases, for choosing one-of-many, the iPhone shows a checkmark by the chosen item (Figure 7.10(d)), harking back to the Macintosh menu bar checkmarks (Figure 7.6).

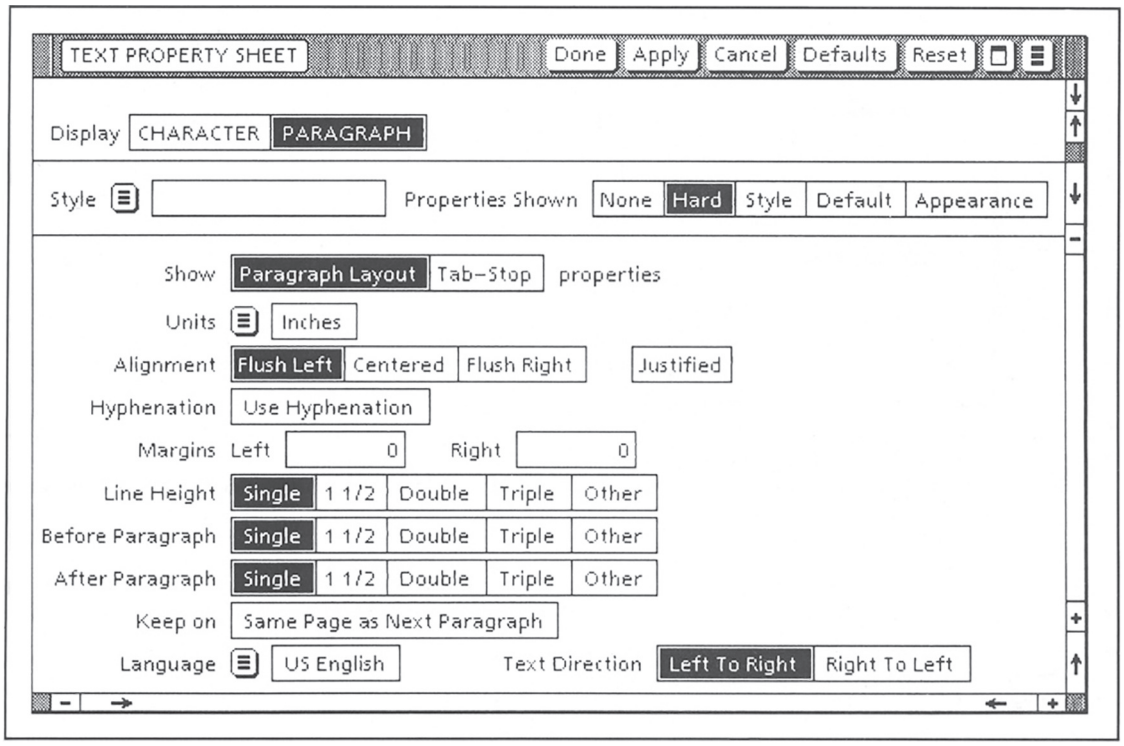


Figure 7.8 Xerox Star property sheet. Source: Fig. 2 of Johnson et al. [1989]; see also http://www.digibarn.com/friends/curbow/star/retrospect/.

Another dimension to the graphic design is the dynamic response to the user's actions—in particular, animations. For example, when the user selected a menu bar subitem on the original Macintosh, that item blinked a few times before the menu disappeared, which made it easier for users to see what happened. This might be useful—for example, if they accidentally released over the wrong item. (The Lisa did not do this.) Newer systems (since around 2001) have incorporated other animations such as fading in and out for the popup menus. Another dynamic effect is providing "tooltips" or "bubble help" that appears when the user hovers the mouse over a menu item (or other widget) without moving. These are covered in Chapter 16.

7.3.8 Default Value?

Many menus can have a *default value*, which is highlighted when the user first uses or sees the menu. Examples of the default value are checkboxes or radio buttons that have a value already set when they appear, to show what will be selected if the

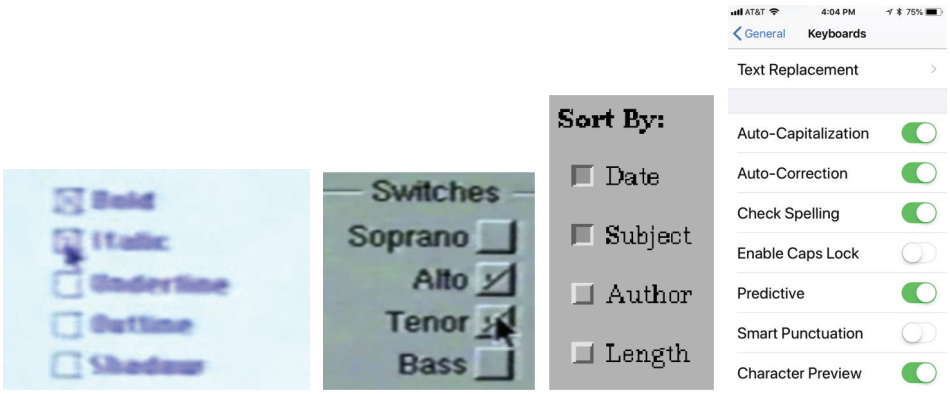


Figure 7.9 Various checkboxes. Sources: (a) and (b) Myers [1990b]. (c) Motif Programming Manual, open source, https://www.oreilly.com/openbook/motif/vol6a/Vol6a_html/ch11.html. (d) iOS screen capture, May 2018.

user doesn't change anything, or else a button (typically the *OK* button) that will be selected if the user hits the Enter keyboard key instead of explicitly selecting an item. In the first case, the default value is just indicated in the usual way, by having the widget already filled in. In the second case, the default button is usually drawn differently than the other buttons (Figure 7.12).

Typically, the value that is the default would be the value for the selected item (e.g., if text is selected, the value shown for the *Bold* checkmark depends on whether the selected text is bold or not). Another criterion is the user's last setting (e.g., the value for two-sided printing may depend on what the user selected the last time printing was done). In other cases, the default is picked by the programmer (e.g.,

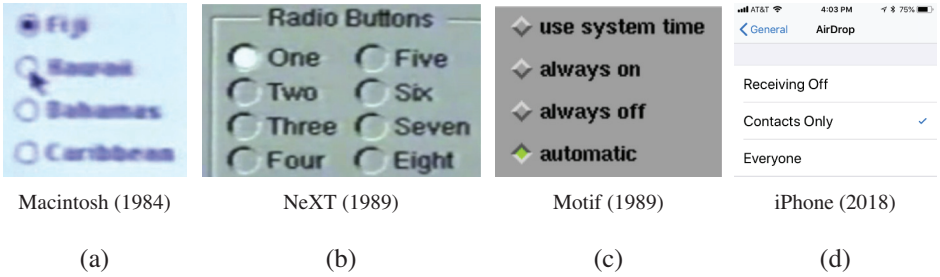


Figure 7.10 Various radio buttons. Sources: (a) and (b) Myers [1990b]. (c) By Thomas Driemeyer [GPL], from Wikimedia Commons. (d) iOS screen capture, May 2018.

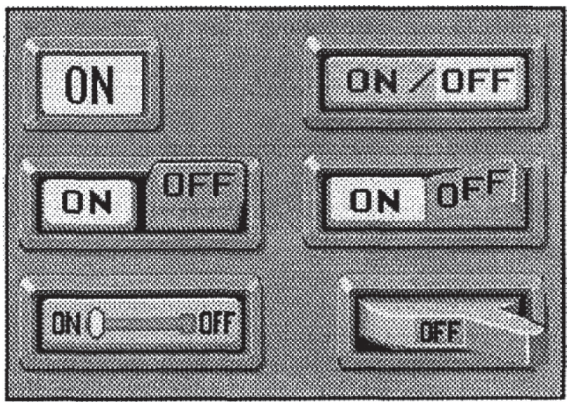


Figure 7.11 Research on different ways for toggle switches to appear on a touchscreen. Source: Fig. 2 of Plaisant and Wallace [1992].

whether *Cancel* or *OK* is the default may depend on the severity of the action that will be performed).

7.3.9 Layout

Most regular menus list the items vertically, whereas checkboxes and radio buttons are often either vertical or horizontal (when horizontal, it is important to provide sufficient spacing so that it is clear which control goes with which label). Buttons can also be horizontally or vertically arranged, but there is no ambiguity about the labels since they are inside the button graphic. Menu bars have the first-level menu horizontal with the rest of the submenus vertical.

The advantages of putting the menu bar at the top of the screen as on the Macintosh, versus on each window as in Microsoft Windows, are that it makes it easy to remember where they are (since they are always in the same place), and it is easier to get the mouse to the correct location. As discussed in Section 4.14.1, the larger the target, the easier it is to hit. The first-level items in the Macintosh menu bars essentially have an infinite vertical size since the mouse will peg at the top of the screen, so the user can shoot the mouse up to the top and does not need to slow down to be able to accurately get to the correct vertical position. Further, the menu bar takes up less room than if there was one needed for each window, a key

Figure 7.12 Original Macintosh Cancel–OK buttons. The thicker border shows that OK is the default. Source: Macintosh screen capture, May 2018.

advantage for the tiny original Lisa and Macintosh screens (Figures 5.11 and 2.16, respectively). However, the distance to move to the top of today's large screens can be significantly more than the distance to the top of a window.

7.3.9.1 Pie Menus and Marking Menus

A special case of layout are *pie menus* (also called *radial menus*) where the items are arranged around a circle, and the items typically popup at the pointer cursor location. A key motivation for pie menus is that all items are an equal distance from the pointer, unlike linear menus where some items are further from the start location. Another motivation is that users can memorize the direction to move to select a particular item, so if there are not too many items users can learn a very quick gesture to select an item that can then be invoked without looking.

An early textbook [Newman and Sproull 1973] identified a system called PIXIE from 1969 with popup radial menus [Wiseman et al. 1969]. They also appeared as "gesture menus" in Fabrik in 1988 [Ludolph et al. 1988]. However, more influential was the extensive research at the University of Maryland exploring many aspects of the pie menu designs (see Chapter 33 for a discussion by Don Hopkins about that era). An early paper [Callahan et al. 1988] showed that eight-item pie menus were about 15% faster to use than the equivalent vertical menu, but it is important for the segments to be in the cardinal directions, as in Figure 7.13(a). Their earliest implementations of pie menus supported not displaying the menu if the user moves fast enough—for example, when the user remembers which direction from the center the desired item is. For my 1990 "All the Widgets" video, Don Hopkins prepared a 2½-min segment that demonstrates many variants [Myers 1990b, at 25:46]. For example, pie submenus may appear when the user clicks where the submenus overlay the parent menus, and the distance from the center may be used to select parameters of the item such as the saturation of a color or the size of a font, thereby combining the menu selection with a slider. Another demonstrated idea is to provide a flexible lever for fine-grain control of the slider value (Figure 7.13(b)). Since users might memorize the direction for menu items, a 1991 article recommends avoiding the delay of popping up the submenu graphics if the user clicks ahead [Hopkins 1991].

Kurtenbach and Buxton [1993] designed a variant of pie menus that they call *marking menu*s, where users did not need to lift their finger off the mouse button (or the stylus or finger off the screen) but could continue from parent menu to submenu just by moving in the correct direction. As users memorized the direction for the items, they could make zigzag marks as a *gesture* to select parent and

(a) (b) (c)

Figure 7.13 (a) Pie menu. (b) Flexible control. (c) Too many items. Sources: (a) Callahan et al. [1988] Fig. 6. (b) and (c) Myers [1990b] at 28:05 and 26:28.

submenu items, which was shown to be about 3.5 times faster [Kurtenbach and Buxton 1993]. If users were unfamiliar with the items at each level, they could pause for approximately 1/3 second, and the pie menu would be displayed.

Whereas both pie menus and marking menus have been used in some research and commercial products (see Chapter 33), there are serious problems that make their widespread application problematic. A key issue is that they take up significantly more screen space than linear menus. In order for the labels to be readable and not overlap, the menu must get bigger if the names are long. This is related to the problem of scaling to a large number of items (Figure 7.13(c)), in which case the part of the circle allocated to each pie piece is quite small near the center, so the user must move further out from the center to have sufficient accuracy, overriding the advantage of the smaller amount of movement required. Another issue is that in practice pie menus work best when there are an even number of items, so users can accurately remember which direction to move, with best results for 2, 4, 6, 8, and 12 items [Kurtenbach and Buxton 1993].

7.3.10 Where the Menu Is Displayed

Usually, menus are displayed on the screen, with the key question being where they are positioned. The Macintosh menu bar is at the top of a screen, whereas the Windows menu bar is at the top of each window. Palette menus are usually at the left of the application but can also be at the top or (rarely) at the right. Checkboxes, radio buttons, and option menus are usually collected into a ribbon or a dialog box.

However, some menus have been displayed in unusual places. For example, menus might be projected onto a wall, and then the user might select the menu item using a laser pointer and a remote camera as in Figure 4.19 (assuming the wall does not have its own sensors). Researchers have investigated projecting menus

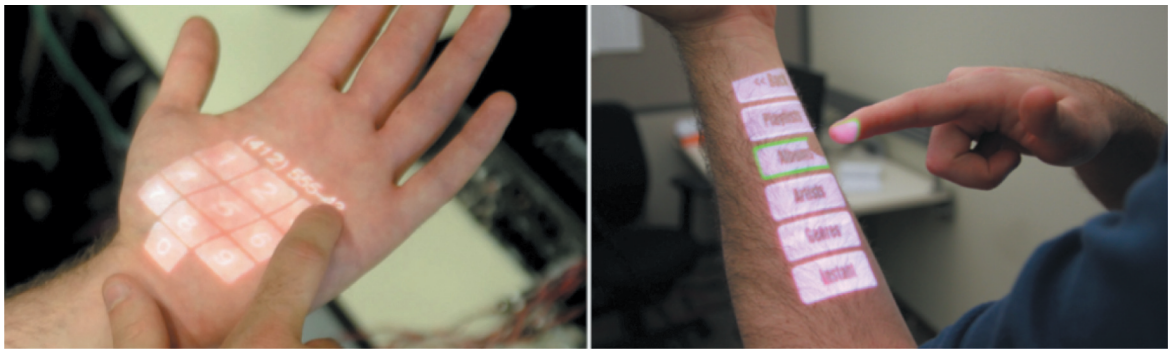


Figure 7.14 Projecting menus on the hand and arm. Source: Excerpt of Fig. 1 from Harrison and Faste [2014].

onto body parts (see Figure 7.14), and detecting where the user taps through body-based sensors, such as the acoustic signal passing through the user's arm from a tap on the hand or arm [Harrison and Faste 2014].

7.3.11 Behaviors

7.3.11.1 How Many States

As shown in Section 1.1 and Figure 1.3, checkboxes and other kinds of menus can be in many states, and the designer must decide what behaviors to support and what kind of feedback to provide so the user is clear about what state it is in. Typically, the widgets might be in the following states, each of which requires its own feedback:

- Valid/invalid (grayed out), also called enabled/disabled: Whether this widget can be operated now.
- Hovered: Whether the pointer cursor is over the item, so it will be operated if the user pushes the pointer button.
- Interim-selected: While the user is in the process of interacting, which item is the user operating on now. For example, while holding the pointer button down and moving around in a menu before releasing, the menu will show the item under the cursor.
- Selected: Whether this item is selected, such as checked for a checkbox widget. Some widgets support *partially selected* (Figure 1.2).
- Keyboard focus: Which item will be affected by keyboard actions like using the arrow keys or the Enter key.

Some of these are independent and some are not. Many widgets can be invalid while either selected or not selected—for example, if the current item is checked

and the user is not allowed to change it. However, widgets do not usually respond to hover events when they are invalid, and most widgets cannot be interim selected or have the keyboard focus while invalid. Similarly, the *feedback* for these may be independent or not. For example, the Windows checkbox as shown in Figures 1.1 through 1.3 uses fill-color, border width, a dotted outline, and a checkmark icon so hover, interim, final, and focus feedbacks all look different. In other cases, the same feedback may be used for both interim and final feedback (like swapping the foreground and background colors such as toggling from black to white).

7.3.11.2 Moving from One Item to Another

Another dimension of the behavior is how a *set* of items is affected. For example, if the user presses the pointer button down on one menu item and moves to another menu item while holding down the button, the interim-selected item will usually change to the new item being pointed to, and whichever item is under the cursor on release is the one that is selected. But what about for a set of checkboxes? Different systems have made different choices. On the Macintosh, only the first pressed checkbox can be manipulated—if you press down on one checkbox and move away from that checkbox and release, then nothing will happen, no matter whether you move into the background or over another checkbox. To operate a different checkbox, you have to start pressing down on top of that checkbox. I have called this a *first-one-only* behavior [Myers et al. 1997b]. However, on the NeXT computer checkboxes operate like menus, and you can move from one item to another while pressing [Myers 1990b].

7.3.11.3 Pointer Operation

Most early menu systems (including the original Interlisp, Macintosh, and Windows designs [Myers 1990b]) popped up a menu on pointer button down and stopped on pointer button up. This might be called the *press-drag-release* behavior for menus. When that menu item had submenus, the user was required to move from the parent menu to the submenu while the mouse button was held down, which may have required moving through multiple levels of submenus before releasing. An alternative design is to support clicking (down and up) on a menu to bring up its submenu, and then clicking again to select the item in the submenu, and so on, which might be called the *click-move-click* behavior. This style was supported by the early Smalltalk system [Myers 1990b] and was provided for many menus in addition to press-drag-release starting around 1987. It is now available on most menus, including those on phones and webpages.

An argument for the press-drag-release style is that it keeps the user in a state of tension until the operation is finished, so the user is less likely to make mode

errors. It is also easier to implement and less prone to bugs—for example, it is common with click-move-click implementations to accidentally leave the menu displayed if the user clicks in an unexpected place, like outside of the active area. On the other hand, click-move-click is much easier for users since it can be difficult to continuously hold down the mouse button or to drag across a touchscreen while touching.

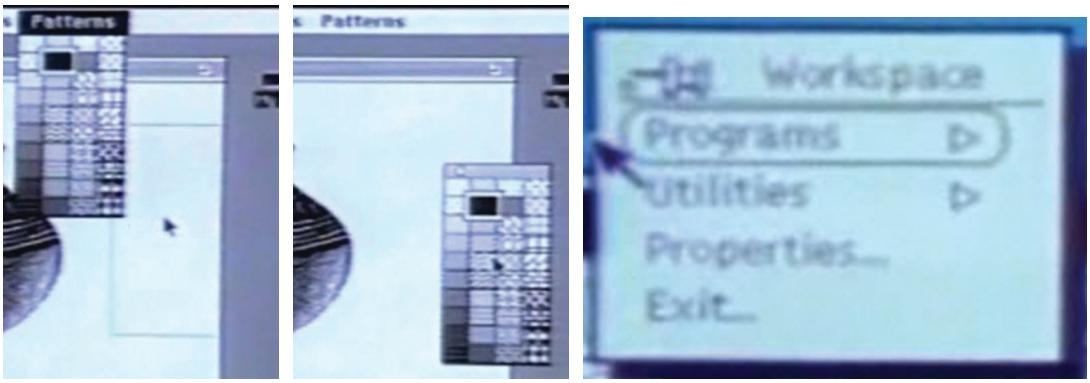
Another consideration affects slide out submenus. Consider the path that the pointer must take—for example, to go from the "Style" to the "Shadow" in Figure 7.4(b). Most early menus required that the cursor move horizontally along the "Style" item to get into the submenu, whereas the normal path for the pointer would be a diagonal movement that would cross "Align Left," thereby removing the "Style" menu [Kobayashi and Igarashi 2003]. This made it frustrating to get the desired submenus, especially when the higher-level menu item was short and wide. Well-designed menus (such as those built into major platforms like Windows or the Macintosh—dating back to the original Macintosh Human Interface Guidelines in 1984, but often not done in hand-built menu systems in web pages) provide a delay before the original submenu is removed, so the user has time to move diagonally into a submenu item. To get a different submenu, the user pauses briefly over the new higher-level menu item if the appropriate submenu isn't displayed.

Another behavior for menus is as *tear-off* or *pinned* menus, where there is a way for users to convert popup menus into fixed menus. In HyperCard one could press down over the menu and move far away, and the menu would tear off from the menu bar and become a window containing a fixed menu, with a close box to get rid of it (Figure 7.15(a) and (b)). On the NeXT, submenus always had title bars that could be used to drag them away from the parent menu. OpenLook provided a *pin* icon on some popup menus (Figure 7.15(c)), which could be clicked to convert the menu into a fixed (pinned) menu [Myers 1990b, at 34:23].

7.3.11.4 Keyboard Operation

The original menus on early computers were all operated with the keyboard, either by typing a number for the desired selection (Figure 7.3) or using keyboard arrow keys. The original GUI menus, like in Smalltalk, could not be operated with the keyboard. On the Lisa and Macintosh, *keyboard shortcuts* (also called *shortcuts*, *hotkeys*, or *accelerator keys*) for menu items were displayed to the right of the relevant items, like "Apple-C" for Copy (Figure 5.11), where the "Apple" key was a special key on the keyboard. Unlabeled menu items could not be invoked from the keyboard.

When Microsoft Windows for the PC was released, it was expected that many PC users would *not* have a mouse, so there needed to be a way to invoke all menu commands. The control key (sometimes combined with other modifier keys, like



(a) (b) (c)

Figure 7.15 Converting a popup into a fixed menu. (a) HyperCard tear-off before and (b) after. (c) OpenLook had a pin icon at the upper left of some menus. Source: Myers [1990c] (a) and (b) at 29:19 and (c) at 34:23.

Ctrl-Shift or Ctrl-Alt) would provide shortcuts like on the Macintosh. In addition, by hitting the "Alt" keyboard key by itself and releasing it, the menus would capture all subsequent keyboard input, and arrow keys could be used to move around the menu items. Also, *every* menu item had a unique letter called the *mnemonics* that would cause it to be invoked. This was shown visibly by underlining it—for example, hitting and releasing Alt and then hitting and releasing "m" would invoke "Move" in Figure 7.16. The original Windows 1.0 (1985) menus did not have marked keyboard shortcuts, but these were added in Windows 2.0, which was released in 1987 (Figure 7.16). Thus, there were three different ways to use the keyboard to select menu items (shortcuts for some commands, using Alt followed by using the arrow keys, or using Alt followed by the marked mnemonic letter).

For menus used to provide values, including menus on web pages, one can usually type the first letter of a value to scroll to that place in the menu. This is particularly useful in menus of country names or US states that tend to be in alphabetical order and long. The user can then use the arrow keys or pointer to move up and down from there, and the Space or Enter key to confirm. Typing the second letter of the name does not work, however, since that will scroll to the items that start with that letter instead (except in a combo box, where the typed letters are put into the text box).

The Palm provided shortcut keys using a diagonal up gesture followed by the gesture for a letter, such as diagonal-up-C for copy (Figure 7.7(a)). PenWindows provided the same shortcuts as regular Windows. Today's smartphones and tablets

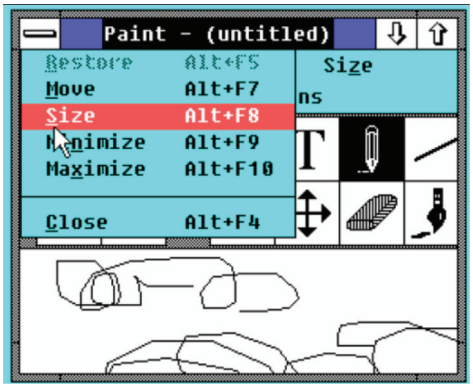


Figure 7.16 Windows 2.0 menu showing the underlined mnemonics (e.g., "M" for Move and "x" for Maximize) and shortcuts (e.g., Alt+F7 for Move). Source: Courtesy Nathan Lineback (ToastyTech), http://toastytech.com/guis/win203.html.

do not have physical keyboards, so there are no keyboard accelerators, but if you attach a keyboard, then the usual shortcuts work—for example, an external keyboard for an iPhone or iPad has an "Apple" key that performs the copy operation on selected text when you hit Apple-C.

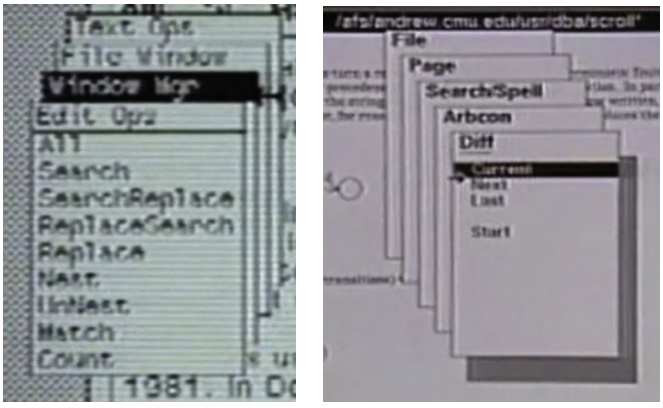
7.4 Examples of Historical Menus

Although the previous sections have covered most of the variants for menus that have been used in various systems, this section summarizes them in chronological order rather than by feature. This section is based on the "All the Widgets" video [Myers 1990b] for older systems, and my research for more recent ones.

- Obviously, physical menus have existed in the real world for hundreds of years, such as the menu in restaurants.
- Consumer electronics, such as car radios, had push-button station selection since the late 1930s (the original *radio buttons*) [Demain 2012]. Also in the late 1930s, television sets started to be available with a choice of 19 channels (rather than the continuous dial of radios), eventually settling on 12 channels in the very high frequency (VHF) band (numbered 2–13) in the late 1940s [Reiser 1982]. These were navigated typically by a rotary dial with the correct number of notches or stops, but remote controls that could move up and down by one channel at a time started appearing in the 1950s.
- Xerox PARC's Markup graphics editor (1975) may be the first system to have popup menus (Figure 2.12(a)). The Alto Draw program (Figure 2.12(b)) was

created around the same time (1975) and used fixed menus and palettes (see Section 2.4.3.3).

- The Xerox Development Environment (XDE) from 1975 explored overlapping submenus in a stack, where the user could switch from one to another by moving left and right (Figure 7.17(a)).
- In 1976, Buxton demonstrated a *moving menu* where the selection point stays fixed and the items move instead [Buxton 1982], a behavior that is similar to that used now in spin menus like on the iPhone (Figure 7.7(b)). A variation on this behavior is a *toolglass* (also called *magic lens*) [Bier et al. 1993] where dragging a widget on top of the target area makes aspects of the underlying object visible (so the toolglass is semi-transparent), and then the user interacts with the underlying object through the toolglass.
- Smalltalk from 1976 (Section 2.4.3.1) made extensive use and popularized popup menus with cascading submenus (Figure 2.10(b)). The first use of radio buttons and checkboxes was in the Smalltalk system. Ted Kaehler in the Smalltalk group coined the term *radio button* based on the buttons on old car radios (see Section 7.2 and Chapter 29).
- The Xerox Star (1981) popularized property sheets (now called dialog boxes) containing choice parameters (Figure 7.8). The Star also provided a menu button using a *hamburger icon* (Figures 5.10(b) and 7.2(a) and Section 26.2).
- The Xerox PARC Cedar (1982) research environment (Figure 11.3) provided fixed menus, buttons, and property sheets at the top of each window.



(a) (b)

Figure 7.17 (a) Xerox XDE stacked menus. (b) The Andrew menu stack. Sources: Myers [1990b] at (a) 18:34 and (b) 24:39.

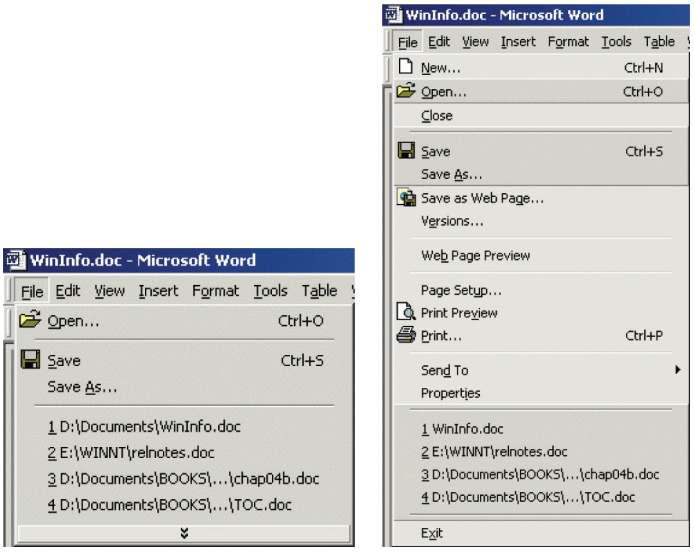
- The Apple Lisa (1983) and Macintosh (1984) introduced menu bars, combo boxes (Figure 5.11), and buttons with optional default actions (Figure 7.12), and also grayed out items. The menu bars in the Macintosh were the first to have submenus, the blinking feedback to show which item was selected, and "..." to signify items that displayed dialog boxes.
- The Symbolics window manager (1985) was one of the first to support keyboard navigation through GUI menus (Section 2.2.2) by typing the menu item or using arrow keys.
- The Xerox Alternate Reality Kit system (Section 2.4.10) provided menus and buttons with a three-dimensional (3D) look through shading (1985) to make the objects on the screen look more real.
- Microsoft Windows 1.0 (1985) and especially Windows 2.0 (1987; Figure 7.16) provided similar widgets to Apple including menu bars, checkboxes, and radio buttons. They also supported keyboard navigation to select any menu item.
- The Andrew system (1985) used a stack of cards for its menus (Figure 7.17(b)) rather than a menu bar.
- Apple's HyperCard (1987; Section 2.4.9) introduced tear off menus that turn into fixed menus and palettes (Figure 7.15(a) and (b)).
- The first-level menu bar menus in OpenLook (1988; Section 2.4.12) looked like option buttons (with down-pointing arrow icons). OpenLook also supported pinned menus (Figure 7.15(c)) and had choice options modeled after the Star in addition to regular checkboxes.
- The NeXT (1989) replaced the menu bar with a vertical fixed menu that could be moved around (see Figure 2.20) and had submenus that could be pinned by dragging them away from the parent.
- The Motif system (1989; Section 2.4.12) had a unique graphical style where checkboxes were squares (Figure 7.9(c)) and radio buttons were diamonds (Figure 7.10(c)).
- A research system from IBM in 1989 demonstrated having icons in the palette be animated to show their actions [Alpert 1991].
- Meanwhile, in the 1980s television sets started using on-screen displays to provide menus for their own options (rather than physical knobs and switches) as well as the menus to control connected devices like videocassette recorder (VCR) recorders or cable set-top boxes.

- The PenPoint system from Go (1991; Section 3.5) used a conventional menu bar (Figure 3.9) that was operated by tapping with the pen.
- Early wireless phones (1993–2003) tended to use menus operated by arrow keys as well as two soft keys operated by buttons just below the screen.
- Palm devices (starting in 1996) used a menu bar at the top of the screen with character gesture accelerators (Figure 7.7(a)) and also buttons, checkboxes, and combo boxes.
- Windows CE (1996) moved the menu bar to the bottom of the screen (Figure 3.21(a)).
- The RIM Blackberry (1999) primarily used a side-mounted physical rocker switch or dial to select menu items (Figure 3.23).
- Microsoft Office XP (2001) introduced adaptive *personalized menus* that by default only showed a *short menu* of what was used most often and could be opened to show all items (Figure 7.18). This feature was widely disliked and luckily was easy to disable so it instead always showed all items.
- The original Apple iPod (2001) used an innovative physical scroll wheel to navigate among the menu items, with a physical button in the center for selection (Section 3.17).
- The iPhone (2007) significantly broke with previous designs. It rarely used menus, checkboxes, or radio buttons, and it provided different widgets for these such as spin menus (Figure 7.7(b)) and switches (Figure 7.9(d)).
- Today, menus in 3D and virtual and augmented reality (VR/AR) have mainly used conventional flat menus superimposed on the scene, but see Chapter 12.

7.5 Research on How to Organize Menu Contents

A key topic about menus that has not yet been discussed is how to organize the *contents* of menus—in what order should the items be listed, and how to break them up into submenus. This has been the topic of hundreds of research papers, and there is even a full 368-page book on this topic [Norman 1991]. Design guidelines, such as Apple's [Apple Computer 1992], cover this topic as well. Clearly, this section cannot cover the entire topic, so I will summarize the recommendations.

The number one requirement is to follow the guidelines for the target platform. For example, for Macintosh interfaces, the contents of the File, Edit, and some other menus are well-defined. Further, the guidelines specify the *names* that must be given to certain operations—such as what commands like Open, Close,



(a) (b)

Figure 7.18 Microsoft Office XP "personalized menus" showed (a) the most frequent or (b) all commands. Sources: Microsoft Office XP screenshots, used with permission by Microsoft.

Delete, Cut, Copy, and so on must do. Other platforms have similar specifications. For items where there are no platform specifications, designers must be careful to give them clear and distinctive names—similar but distinct operations should be clearly distinguishable from their names. The Apple guidelines recommend:

> For menu items that act as commands, use verbs (or verb phrases) that declare the action that occurs when the user chooses the item. ... If the item changes an attribute of a selected object, use an adjective (or adjective phrase) that describes the change. Adjectives in menus imply an action. They should fit into the sentence "Change the selected object or objects to ..." ... Use one word for menu item names when possible. Capitalize the first letter of the first word of each command and capitalize the important words in phrases. [Apple Computer 1992, p. 58–59]

A classic problem arises when there are two choices. Using two menu items or two buttons can be confusing to users as to whether they describe the current state or what will happen. For example, the Macintosh guidelines warn that a menu item or a button labeled "Grid On" is ambiguous [Apple Computer 1992, p. 76]. This problem still exists today, as identified in confusing switches in Windows 10

[Whitenton 2018]. A better design is to use a single item with a checkbox ("Grid") or two opposing items with radio buttons ("Grid On," "Grid Off").

With respect to the organization of menu items in a menu, the key recommendation is to group them in a logical order for users. A logical hierarchy was shown to be less error-prone and faster than a random or alphabetical order in multiple studies [McDonald et al. 1983, Norman 1991]. If there is no logical order or for lists of items of the same type (such as fonts or countries), then alphabetical order should be used. It is also recommended to put the most commonly used items at the top: for example, the top of the Edit menu should have the most popular commands. If there are a large number of items, a split menu has been shown to work well [Sears and Shneiderman 1994]. For example, modern font menus have a few commonly used items at the top, followed by all options in alphabetical order (see Figure 7.2(d)). Most systems provide separators, which should be used when there are different groups of items in the same menu.

If there are submenus, studies of conventional menus as well as web pages have shown that it works better to have wider and shallower menu trees rather than narrower and deeper trees [Norman 1991, Larson and Czerwinski 1998, Shneiderman et al. 2017, p. 258–260]. That is, each level should have more items so there can be fewer levels. In fact, the Macintosh guidelines require there to be only one level of pull-out menus in a menu bar (i.e., the top-level horizontal menu bar items have pull-down submenus, but those submenus should only have further cascading submenus when necessary, and only one more level—so there are no more than three levels total [Apple Computer 1992, p. 82]).

Most of these recommendations apply equally well to the names and organization for buttons, checkboxes, radio buttons, and other elements.

7.6 Summary of Design Considerations for Menus

There are a variety of design considerations for menus. First, considerations for a designer selecting which kind of menu to use in a user interface:

- How many items need to be displayed?
- How many items need to be selected at once?
- Hierarchical or flat organizations for the items?
- Role of the items to be selected among—are they commands or values? Do they control modes (like the drawing mode when a palette might be preferred)?
- Does the menu need to display the current value or a default value?

- Does the menu need to indicate whether some items are not valid to be selected now?
- Should the menu item be able to be selected using a keyboard and/or arrow keys, in addition to a pointing device?
- What kind of menu to use (which will be influenced by the above choices)?
- What graphic design will best fit in with the rest of the user interface? Can a default look and feel just be used?
- How will the items of the menu be ordered (e.g., alphabetical, by topic) and are separators displayed between groups?

Next, additional low-level decisions that might be relevant to a designer creating a new kind of menu or implementing a menu from scratch:

- How many states need to be supported? For example, does the menu need to support hover, interim-selected, and/or final-selected? Does the menu need to show if it has the keyboard focus?
- What layout to use for the menu—for example, pie menus versus linear menus, and where will submenus be displayed?
- What happens when the pointer moves from one item to the other, especially if this is done while the mouse button is pressed? (For example, does the selection move from item to item like in a regular menu versus just go on and off on the first item pressed over, like a checkbox panel?)
- What is the detailed interaction with the pointer? For example, can the menu be operated by clicking, or is press-drag-release required, or are both supported?
- What is the detailed interaction with the keyboard? For example, can the user type the first letter of items? What about the second letter? Can the arrow keys be used to navigate the menus? How is the selected item triggered from the keyboard (usually, the Enter key)? Is there a shortcut that will invoke an item with a single keystroke (like Ctrl-C for the Copy item)?
- For submenus, how are they triggered, and how does the user move from the parent menu to submenus (and back)? How accurate must this movement be to avoid the top-level item becoming unselected?

7.7 Chapter Conclusions

Menus of all kinds are still a key element of computerized devices and will remain important for the foreseeable future. It is easier for users to pick from a set of

visible choices rather than having to generate the desired choice, and menus have the advantage of being able to display the current options and current selection(s). Interactions that make it easier for users to select items from menus on tiny devices (like watches) or enormous screens (like televisions at a distance) are still needed, as well as menu interactions for multi-modal situations like talking while looking at menu items on a screen and for selections in AR/VR.

CHAPTER 8

Text Entry and Editing for Computers and Handhelds

8.1 Introduction

Text entry and text editing has always been an important and well-studied area of interaction techniques. *Text entry* is the process where a user inputs characters into a computerized device. And *characters* mean any of the elements the user types, including letters, numbers, punctuation, and so on. In Western languages other than English, characters may include additional marks like accents. In Chinese (Section 8.2.2) and similar languages, multiple strokes are often needed to enter a single character.

For early computers up through today's laptops, text entry has always been through keyboards (Section 8.2). However, with the advent of mobile devices (see Chapter 3), there has been significant research and commercial exploration of how to do text entry for those (Section 8.3). This work continues today since (I claim) there is still not a satisfactory way to enter large quantities of text into a mobile device. As evidence, consider that many people are willing to read a long book on their phone, but virtually no author would be willing to try to *write* a book using only a phone. And there still isn't any good way to enter text on a computerized watch. Note that text entry using voice transcription is deferred until Chapter 18 as it is usually combined with using speech for giving commands.

Given that text entry techniques differ significantly in terms of the speed of entry, there has been much research on the appropriate way to measure this speed (typically *words per minute* or *wpm*). Whereas it may at first seem like one only needs to count the characters entered in a period of time, this is complicated by the issue of errors—a method where one can enter lots of incorrect characters quickly is of little use compared to a slower but more accurate method. Section 8.5 discusses appropriate ways to measure the speed and error rate for text entry methods.

Text editing is the process where a user can change the text that is already in a computerized device (Section 8.6). Text editing includes deleting, changing, moving, and entering text and involves positioning the text cursor and selecting text to be modified. Also relevant is *text formatting*, which involves controlling how the text looks on the screen or on paper, such as selecting fonts, bold/italic, and positioning. A final topic for this chapter is *text entry widgets*, such as single-line and multi-line text input fields (Section 8.7).

8.2 Text Entry for Desktops

Using the distinctions defined in Chapters 2 and 3, this section discusses text entry for desktop and laptop machines while Section 8.3 will cover mobile devices.

8.2.1 Typewriters

The *keyboard* used with today's computers is based on the keyboard first invented for mechanical typewriters. Inventors had been investigating how to enable faster typing for years, until the first commercially successful typewriter was developed by Christopher Latham Sholes, Carlos Glidden, and others in about 1868 (see Figure 8.1). The patent on this device was sold to the Remington company, which produced it commercially. This device used the QWERTY layout of keys to reduce the likelihood of keys hitting each other and jamming. Various innovations and small changes were made over time, eventually leading to the QWERTY layout

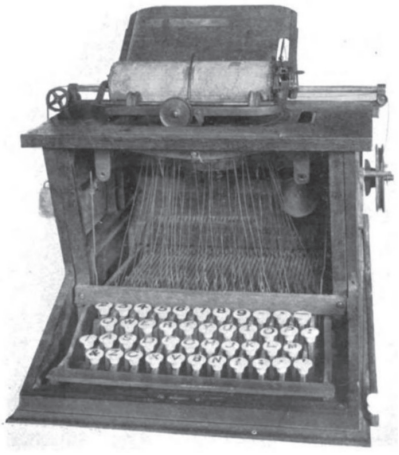


Figure 8.1 Sholes typewriter, 1873. Source: George Iles (1912) Leading American Inventors, Henry Holt & Co., New York, USA, p. 328 on Google Books, Public Domain, https://commons.wikimedia.org/w/index.php?curid=3361647.

being used in the terminals for the early computers (see, e.g., Figures 2.3 and 2.9(b)).

There have been a number of variants of key layouts used over the years, with the most famous being the Dvorak Simplified Keyboard layout. This was invented in 1936 by Dr. August Dvorak with the goal of increasing the typing speed by taking into account the letter frequencies in English and the physiology of people's hands. Although this layout is an ANSI standard, it is rarely used.

Today, the QWERTY format invented by Sholes is essentially the only layout used for English, with small variants used for other Western languages. Typical text entry speeds for a QWERTY keyboard range from hunt-and-peck (two fingers) typing at about 27–37 wpm, to 35 wpm for moderate typists who know how to *touch type* using all fingers, to professional typists at 50–80 wpm, up to 120 wpm. The fastest recorded speed on a regular typewriter is over 200 wpm (see Table 8.1).

8.2.2 Chinese Text Entry[1]

Chinese is one of the most challenging languages to learn in the world. The challenge comes from multiple aspects including the complex composition of characters, the large number of different characters that need to be memorized, and the fact that knowing how to write it does not help with how to read it. It is also hard to develop a way to express it in a 13-inch space so it can be typed, and there have been many different Chinese text entry techniques for computing systems over the years.

8.2.2.1 The Chinese Language

Chinese words can be composed of one, two (usually), or more *characters* from the thousands of Chinese characters. Each character can be dissected into *radicals*—a part of the character that is composed of one or more *strokes*. Radicals are much more limited than characters, as the combinations of radicals can compose different characters (see Figure 8.2). Although it is impossible to put the thousands of Chinese characters on a keyboard, there are multiple ways to use the radicals and strokes as a way to type Chinese.

There are no capital letters in Chinese, and characters are written in the same form regardless of their position in the sentence. Sentences are split by punctuation marks according to the nature of the sentences, which is similar to English. However, there are no spaces between different words and sentences.

1. This section is based on the final project research from the IxT class in 2022 by Auli Shen, Sarah Chen, Jeni Huang, and Junqi Wang. Reused with their permission.

Table 8.1 **Speeds for various text entry methods in this chapter. The wpm are given for different levels**

wpm	Text Entry Method	Level (beginner, expert, etc.)	Citation
27–37 35 50–80 200	QWERTY typewriter (Section 8.2.1)	• Two-finger hunt-and-peck • Moderate touch type • Professional typist • Fastest recorded	
180–225 375	Stenographers chorded keypad (Section 8.2.3)	• Typical • World record speed	
6 50–60	EdgeWrite (Section 8.3.2)	• Novice • Word-completion version	Wobbrock et al. [2003] Wobbrock et al. [2006]
10–12 21	Multi-tap on keypad (Section 8.3.3.1)	• Average • Experts with much practice	MacKenzie and Soukoreff [2002b]
15 40	T9 on keypad (Section 8.3.3.2)	• Novices • Experts	MacKenzie and Soukoreff [2002b]
30 60	Small physical keyboard (e.g., RIM Blackberry) Section 8.3.4	• First session • 20th session	Clarkson et al. [2005]
4.3 26 47 67	Twiddler handheld chord device (Section 8.3.5)	• Novices • 400 minutes of practice • 25 hours of practice • Best recorded speed	Lyons et al. [2004]
1	Scanning keyboard (Section 8.3.6)	• Severe disability	Damper [1984]
58.2	OPTI virtual keyboard (Section 8.3.6.1)	• Predicted by model	MacKenzie and Zhang [1999]
40	Metropolis virtual (Section 8.3.6.1)	• Predicted by model	Zhai et al. [2000]
20	Cirrin (Section 8.3.6.1)	• Author after 2 months	Mankoff and Abowd [1998]
9.3	ZoomBoard on watch (Section 8.3.7)	• After eight trials	Oney et al. [2013]
22 24	WatchWriter (Section 8.3.7)	• Tapping • Gesturing	Gordon et al. [2016]
15	Watchboard (Section 8.3.7)	• After lots of practice	Lockerman et al. [2017]
35	Tap physical device (Section 8.3.7)	• 20 days of TapAcademy	Tap Systems Inc. [2019]
8	POBox autocomplete (Section 8.4.4)	• Novice	Masui [1998]
54 88	iPhone using two thumbs	• The author • Fastest person	Evans [2010]
100–150	Normal speaking speed		WordCounter [2016]
360	Fastest intelligible speech	• "Hyperactive radio announcer"	Fields [2010]
14 (5–23)	Average handwriting speed (range)	• Adult	Bledsoe Jr [2011]
60–80 350	Shorthand	• Typical • World record speed	New York Times [1922]
90–142	Signing with ASL (Section 14.3.1.1)	• Typical fundamental rates	Al-khazraji et al. [2018]
238 (175–300) 260 (200–320)	Normal reading rates—ave. (range)	• Non-fiction • Fiction	Brysbaert [2019]

Figure 8.2 Composition of Chinese and English writing from the smallest units. Source: Auli Shen, Sarah Chen, Jeni Huang, and Junqi Wang. Reused with their permission.

Written Chinese has two sets of characters: simplified characters and traditional characters. They look similar, with some characters being exactly the same, and the pronunciation is the same. Simplified characters were introduced by the Chinese government in the 20th century to improve literacy in China—they are much easier to read and write. Before the unification of characters by the Chinese government, Chinese characters (and in the form of traditional characters) were written from right to left. As simplified Chinese was introduced, the Chinese government changed the writing to the left to right, regardless of which kind of character is used, to better accommodate international conventions.

Unlike the alphabet in English, the Chinese characters convey meaning rather than pronunciation. For example, when it comes to some characters with the simplest forms, the character sun "日" looks like a rising sun while the character moon "月" looks like a curving moon. For complex characters like mother "妈妈", the radical on the left is exactly woman "女"—which makes sense since mother is a woman. Generally, the character more or less indicates the meaning or the visual.

To better spread the language to the rest of the world, as well as educate more people to utilize the language, in the 1950s the Chinese government introduced a Romanized pronunciation system known as *pinyin* (Figure 8.3). Pinyin makes it possible to use the pronunciation that is very similar to English alphabets (and using the system of letters in English alphabets) to pronounce Chinese. If an unfamiliar character is labeled in pinyin, it can be pronounced easily. When represented

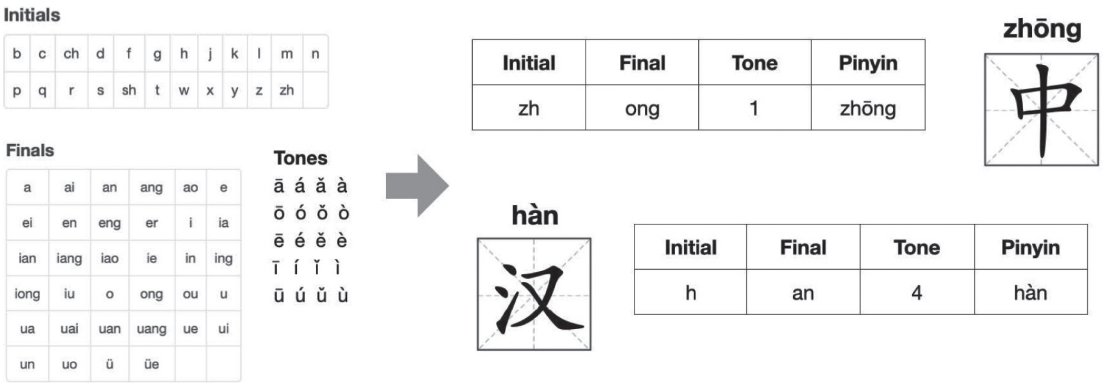


Figure 8.3 The pinyin system. Source: Auli Shen, Sarah Chen, Jeni Huang, and Junqi Wang. Reused with their permission.

by pinyin, each sound in the Chinese language is made up of three parts: initials, finals, and tones. Initials are the first part of a Chinese sound, and finals are the end part of a sound. Tone marks are drawn right above letters when words are written with pinyin. This set of systems indicates another possible way we can type Chinese—since pinyin inherently is composed using the English alphabet, we can utilize the keyboard of English to type Chinese. However, though each character can be uniquely represented by a set of pinyin, the same set of pinyin can represent *multiple* characters. In other words, it is prevalent that different characters are pronounced exactly the same.

8.2.2.2 Chinese Keyboard Typing Methods

Attempts at creating a typewriter for Chinese date back to at least 1916, which used a revolving cylinder containing the characters, ordered by radicals and the number of strokes on the cylinder [New York Times 1916]. After pinyin was standardized in the 1950s, it became much easier to create a way to type Chinese since the regular English keyboard could be used. The Zhuyin method is also based on the pronunciation but encodes the sounds with 37 parts of Chinese characters, rather than using English letters, each of which can be displayed on a key (Figure 8.4).

Other techniques have also been developed. Some are based on the *shape* of the characters rather than their pronunciation. These are more efficient than phonetic input for experienced users because there is no need to select between characters with the same pronunciation, but they can be harder to learn. One is Wubi, where each character is broken down into components that are usually (but not always) the same as radicals. These are typed in the order in which they would be written

Figure 8.4 A comparison of Chinese characters in Zhuyin and pinyin forms. Source: Auli Shen, Sarah Chen, Jeni Huang, and Junqi Wang. Reused with their permission.

by hand. To ensure that extremely complex characters do not require an inordinate number of keystrokes, any character containing more than four components is entered by typing the first three components written, followed by the last. In this way, each character's data can be entered with no more than four keystrokes. An alternative shape-based method is Cangjie, which is popular in Chinese communities that use traditional Chinese characters, like Hong Kong, Malaysia, Singapore, and Taiwan. In Cangjie, each of the radicals is assigned to a key on the QWERTY keyboard, but it does not follow the conventional Chinese handwriting order. It also has a complicated mapping of punctuation marks because rather than using the punctuation mark keys on the keyboard it uses predefined codes that begin with ZX followed by a string of three letters. As a result, it is more complicated and less efficient than Wubi. There are other encodings as well.

Pinyin is the primary method taught in schools in mainland China, so it is the one most commonly used there.

8.2.2.3 Chinese Text Input Interaction Techniques

When using a QWERTY keyboard to input Chinese in a modern computer system, a set of key combinations usually results in a list of possibilities, which requires the input to provide a design for the user to pick the designated character or word from the list. This is especially common for phonetic-based techniques. To reduce the time spent during character selection, intelligent language-based autocompletion, auto-correction, fuzzy match, and word prediction have become an essential part of every Chinese input product.

On a desktop, Chinese input uses a popup component hovering near the text input area but not blocking it. Before selecting a character/word, entered letters will

be displayed with underlines and separated with space based on potential selections. Entering a new letter or deleting a letter will trigger a new set of possible candidates. As shown in Figure 8.5, there is a dropdown containing many more options. Hitting space confirms the selected choice and enters it at the cursor.

On mobile devices, the list of Chinese options replaces the auto-fill bar above the virtual keyboard (Figure 8.6).

Figure 8.5 MacOS Chinese pinyin text entry popup. Source: Screenshot by Auli Shen, Sarah Chen, Jeni Huang, and Junqi Wang. Reused with their permission.

Figure 8.6 iOS 15 pinyin input interface status after typing. Source: Screenshot by Auli Shen, Sarah Chen, Jeni Huang, and Junqi Wang. Reused with their permission.

8.2.3 Chorded Text Entry

Another line of inventions was directed at techniques for taking down exactly what was being said in real-time, eventually creating chorded keyboards for *stenographers* called a *stenotype* machine. The current design dates back to the 1830s and was getting into general use after the 1880s [Stenograph L.L.C 2022]. In order to be faster than regular typing, users hold down multiple keys simultaneously to enter words through their sounds, so whole syllables are entered at the same time. This requires significant training and practice, but professional stenographers generally reach speeds of 180 to 225 wpm, with the world record being over 375 wpm (see Table 8.1).

When Doug Engelbart was inventing the mouse as part of the NLS system around 1968 (Section 2.4.2), he included a regular keyboard, but he wanted a text entry technique that could be used with the left hand while using the mouse in the right hand, and so he also invented a five-button keypad (Figure 2.9(b)). It allowed $2^5 - 1 = 31$ values to be entered, which was enough for the regular letters. Although this device was transferred to the Xerox PARC Alto (Section 2.4.3 and Figure 1.9) along with the mouse, it never caught on.

See also Section 8.3.5 for a discussion of the Twiddler chording typing device for mobile text entry.

8.2.4 Other Text Entry for Desktops

There has of course been research on other ways to do text entry for regular computers. As shown in Figure 4.11, the RAND tablet from 1964 was used to develop early hand printing recognition algorithms for text entry for computers (see Sections 2.5.1.11 and 14.3.1). However, hand printing and handwriting research moved to mobile devices, so they are further covered in Section 8.3. Other approaches have looked at various ways to use the screen and mouse to select characters (e.g., Damper [1984] and Ward et al. [2000]). These are invariably significantly slower than using a keyboard for people without disabilities—but can be advantageous or necessary for people with physical disabilities—and are covered in Section 17.4.3. Similarly, deaf people may communicate with each other using sign language, which can be detected by a camera (with output using an animated avatar on a screen) to provide a high-speed text entry method (about 90–142 wpm, see Section 14.3.1.1).

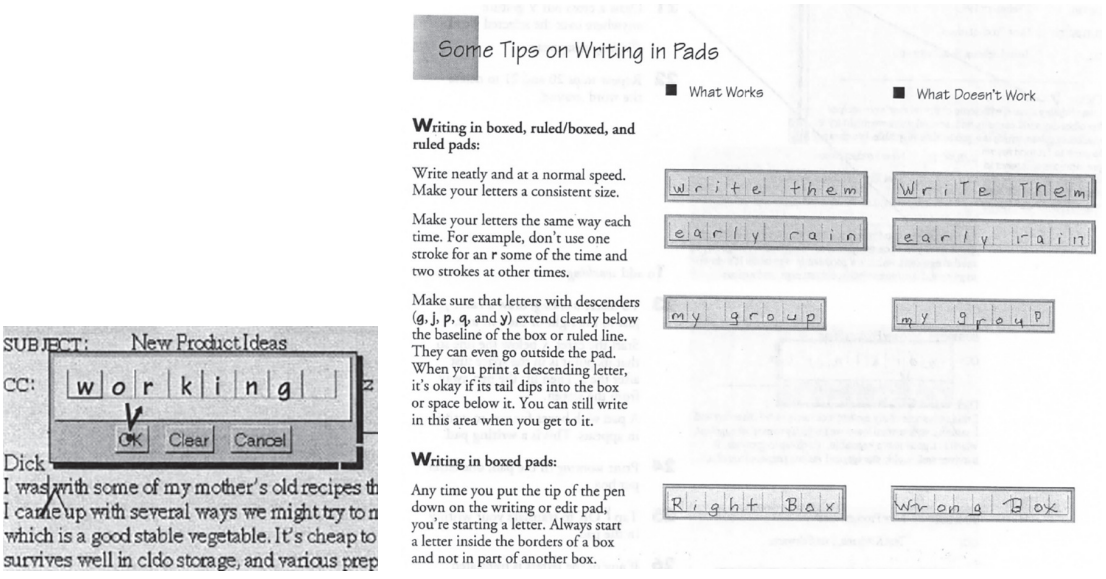
8.3 Text Entry for Mobile Devices

Text entry for mobile devices has had much more variety and research but still remains problematic. A key difference from desktop machines is that most mobile

devices are not expected to be used for large quantities of text, so slower input techniques are considered acceptable. For instance, in a small study in 2016 of college student smartphone text entry, we found that they mostly entered text into map/navigation apps, instant messaging apps, and search engines, whereas they never or seldom entered data into spreadsheets, text editing apps, or note-taking apps [Li and Myers 2021]. They only entered text in 13.0% of their smartphone sessions, and only entered an average of 23.6 characters in each of those text entry sessions [Li and Myers 2021].

8.3.1 Hand Printing and Handwriting

Since the early mobile devices were intended to be like notepads (see Section 3.2.4), it is not surprising that the original plan was to focus on text entry using handwriting. It was quickly recognized that this was a difficult problem, so early systems supported hand printing instead. For instance, Go's PenPoint OS, released in 1991 (Section 3.5), popped up boxes in which users could print individual characters (Figure 8.7(a)). The goal was that people would draw the characters the way they already knew how, so there would be little learning required. However, there was a long list of tips to help the system recognize characters correctly (Figure 8.7(b)).



(a) (b)

Figure 8.7 (a) and (b) Pages from the PenPoint user's guide showing the hand printing boxes. Sources: Go Corporation [1992, pp. 9–10].

Hand printing is fairly slow, so the goal has always been to make entry more "natural" by supporting *handwriting*. Microsoft Pen Windows from 1991 (Section 3.7) supported a popup panel in which users could handwrite (Figure 3.10(b)). Microsoft continued to support Pen versions of their operating systems, culminating in the TabletPC from 2002 that focused on handwriting as the main text entry technique. While much better than earlier recognizers, it still made many errors for most people, and research showed that the user interface provided for correcting recognition errors needed improvement [Shilman et al. 2006].

The most notorious use of handwriting input was on the Apple Newton (Section 3.8 and Figure 3.11), released in 1993, which was not very good and was parodied for a week by Doonesbury (Figure 3.12).

It probably isn't surprising that computerized handwriting recognition is problematic since many people cannot even read their *own* handwriting, and much of the recognition requires inferring words from context since letters are often missing or only hinted at in what is actually written.

8.3.2 Gestural Alphabets

In order to avoid all the problems of handwriting recognition and still allow faster and more accurate text entry using a stylus on mobile devices than is possible with hand printing, researchers and commercial companies investigated using gestural alphabets (Section 3.2.5) that simplified the strokes users needed to make.

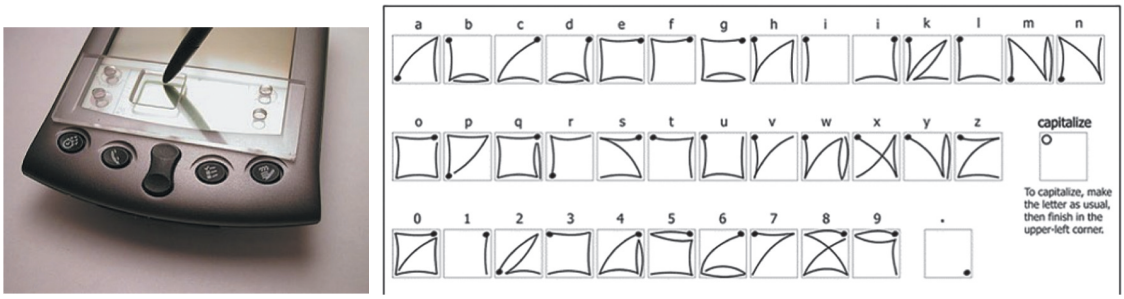
An important early system was *Unistrokes* designed for the Xerox Tab palm-size computer (Section 3.4). Unistrokes [Goldberg and Richardson 1993] had two key innovations. First, it used a gesture set where each letter was drawn with a single stroke. This is in contrast to printing regular letters, many of which require two or more strokes. The second innovation was that the characters could be written on top of each other instead of in separate boxes, as in PenPoint (Figure 8.7(a)). This was enabled since the system no longer had the *segmentation problem* of identifying the end of one character and the beginning of another due to strokes being separated by lifting the pen. Unistrokes sought to further increase the text entry speed and accuracy by making the gestures as simple and distinguishable as possible and using the simplest input for the most frequent characters. Since the characters were recognized as they are drawn, the system could utilize the *direction* in which the stroke was made, even though the result would look the same on paper (e.g., an up stroke vs. a down stroke look the same afterward but while being entered are easy to differentiate). Figure 8.8 shows the amount of time to enter each letter, which was designed to be shortest for the most common letters. This method of input was patented [Goldberg 1993] and published in an academic conference [Goldberg and Richardson 1993].

Figure 8.8 Unistrokes gestures for each character and the median time in milliseconds spent drawing each unistroke. Source: Goldberg and Richardson [1993].

A few years later, around 1995, Jeff Hawkins saw Unistrokes and decided to adapt it to use on his new Palm Pilot device. The Palm version kept the Unistrokes concept (except for "X") and drawing all characters on top of one another, and Palm called it *Graffiti*. A key change was to make the characters much more learnable and recognizable, so the shapes tried to match the way the upper- or lowercase letter was made. For example, the Graffiti "e" could be drawn like a backward "3" so it looked like an upper case "E" (see Figure 3.18(b)), which took much longer to draw than Unistroke's line to the left (Figure 8.8) but was much easier to learn and remember (many characters supported multiple alternative ways to enter them, further increasing guessability). As described above (Section 3.13), Graffiti used the same strokes for some letters and numerals, differentiated by where they were drawn (see Figure 3.18). Punctuation and international characters could be entered after a mode-switching prefix stroke. Since Xerox had a patent on Unistrokes, Palm was sued for copying the design, and Palm had to pay $22.5 million to Xerox to license the technology [Smith 2006]. The Windows CE competitor to Graffiti was called *Jot*.

Although Graffiti and Palm had a brief period of popularity, Palm eventually moved to small physical keyboards for text entry (Figure 3.20(b)), and gestural input fell out of favor until gestures over keyboards came in (described below in Section 8.3.6.2).

Jacob Wobbrock and I developed a gestural input technique called *EdgeWrite* that used the corners of a square to provide additional stability for people whose physical disabilities made it difficult or impossible to use regular gestural alphabets or the tiny keyboard on PDAs or smartphones (see Figure 8.9(a)) [Wobbrock et al. 2003]. Like Graffiti, EdgeWrite's characters were designed to look like the



(a) (b)

Figure 8.9 (a) With EdgeWrite, users move along edges and to the corners of a plastic guide with a square hole mounted on a Palm PDA, to provide additional stability and accuracy. (b) The EdgeWrite gestural alphabet. Sources: (a) Fig. 1 and (b) an excerpt from Fig. 3 of Wobbrock et al. [2003].

regular letters for ease-of-learning (see Figure 8.9(b)), and there was also a word-completion version for increased speed [Wobbrock et al. 2006]. We measured about 6 wpm as initial novice speeds (which was similar to Graffiti) [Wobbrock et al. 2003], going up to 50–60 wpm for the word-completion version [Wobbrock et al. 2006]. More relevant is that a study of four subjects with motor impairments revealed that some were unable to do Graffiti but all could do EdgeWrite. Those who could do both methods had dramatically better accuracy with EdgeWrite [Wobbrock et al. 2003].

Other kinds of gestural text entry are covered later, including gestures made over virtual keyboards, like ShapeWriter and Swype in Section 8.3.6.2, and recognizing sign language gestures in the air, covered in Section 14.3.1.1.

8.3.3 Keypads

Another way to enter text on mobile devices is to use the 12-button keypad that used to be part of all telephones. A *keypad* is an input device that provides a reduced number of keys or buttons. For phones, keypads typically have ten numerals plus "∗" and "#" along with a small number of other keys or buttons. Keypads can also be found on a wide variety of consumer electronic devices, such as copiers, microwaves, and most remote controls (e.g., Figure 4.4(b)), wherever number entry is needed.

With respect to text entry (as opposed to just entering numbers), we need to go back in time to approximately 1919 when the AT&T Bell telephone company introduced *dialing* a phone number. In order to make it easier for people to remember phone numbers, the first three digits were a mnemonic for the neighborhood,

and used letters associated with each number (see Figure 8.10(a)). For example, all phone numbers for my university start with 268- which is "CMU". Note that the "1" and "0" do not have letters because those numbers cannot be used to start a phone number, and "Q" and "Z" are missing, presumably because neighborhood names rarely used those letters.

When push buttons replaced dials on wired phones starting in the 1960s, the same letter associations were retained (Figure 8.10(b)), and these were copied again to mobile phone keypads (Figures 8.10(c) and 3.14) but with Q and Z added back in. Note that the number order for the keypads is *opposite* of the order for cash registers and the numberpads on regular keyboards (where 1 is at the bottom-left). This is because human factors research at AT&T's Bell Labs showed that both arrangements had the same speed, but the 1-at-the-top layout was preferred by the participants.

8.3.3.1 Multi-tap

When mobile phones began to be used for other functions, such as having a local address book which required entering people's names, and sending textual messages, it became important for people to be able to enter text using the keypads. Using the letters that were already on the digits was a natural way to support this.

The simplest way to let users indicate which letter was desired was called *multi-tap*, which, as the name implies, relied on users to tap on keys multiple times. For example, hitting the "2" key one time gives "A", two times gave "B", and three

(a) (b) (c)

Figure 8.10 (a) Dial telephone showing letters associated with the numbers. (b) Conventional touch-tone telephone from AT&T. (c) Close up of keypad for a mobile phone. Sources: (a) By Dhscommtech, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=19589521. (b) Photo by Dante Alighieri at https://commons.wikimedia.org/wiki/File:ATTtelephone-large.jpg. (c) By Sakurambo—Created using Adobe Illustrator CS2, Public Domain, https://commons.wikimedia.org/w/index.php?curid=2048341.

times gave "C". Thus, to enter the word "BET", the user would hit "22338" (refer to Figure 8.10(c)). However, what happens if one wanted the word "CAB", which uses only the "2" key? In this case, there was a time-out, so the user would hit: 222(wait)2(wait)22. If digits were desired, often these would take an extra tap, so entering the number "2" would require hitting the 2 key four times. Special characters were put on the other keys, as shown in Figure 8.10(c), where "0" was "space," "#" was to shift between upper- and lowercase, and "∗" was multi-tapped to enter various kinds of punctuation. Sometimes, an extra key around the keypad signified "enter" or "next" so the user wouldn't have to wait. Note that the keypad layout was not optimized for letter frequencies since frequent letters like "e" and "i" require multiple taps, whereas infrequently used letters like "w" only take one. But it was easy to learn and remember.

Researchers were interested in understanding how fast text could be entered (see also Section 8.5 below), and Scott MacKenzie introduced the concept of the average number of *keystrokes per character* (KSPC) in 1998 as one measure, and found that multi-tap takes 2.03 key hits on average when entering uppercase English words [MacKenzie and Soukoreff 2002b] (a regular computer keyboard is 1 KSPC for letters and numbers, and a little more if the Shift key is taken into account). Mackenzie also measured the average time it takes to enter text, identifying the average rate as around 10–12 wpm, up to 21.0 wpm for experts with lots of practice [MacKenzie and Soukoreff 2002b] (compared to around 40 wpm for regular keyboards)—see Table 8.1.

8.3.3.2 T9

Recognizing the slow speed of multi-tap, the Tegic Company (which was purchased by Nuance in 2007—and Nuance was bought by Microsoft in 2021) invented a text entry technique called *T9*. Tegic's patent on this technology was filed in 1995 and issued in 1998 [Grover et al. 1995] and was widely licensed by almost all mobile phone manufacturers that had keypads. T9 was a *predictive-text* method, which used a language model to guess which word the user wanted based on a single tap of each key. Thus, continuing the example from the previous section, to enter "BET" with T9, you just hit 238. Note that 238 also could be "AFT". If the word you wanted was the predicted one, then you could just move on, but otherwise you needed to cycle through the possible words for those letters, usually using the "∗" key. T9 automatically adapted to the user, so eventually whatever word was selected most would be preferred. Later versions of T9 also had "smart punctuation" where it would predict which punctuation was needed when the user hit the "1" key. The predicted word would be continuously shown, so the whole word changed as more

keys were hit. Phones with T9 also had a way to revert to multi-tap, which would be needed for proper names and other input that wasn't in its dictionary.

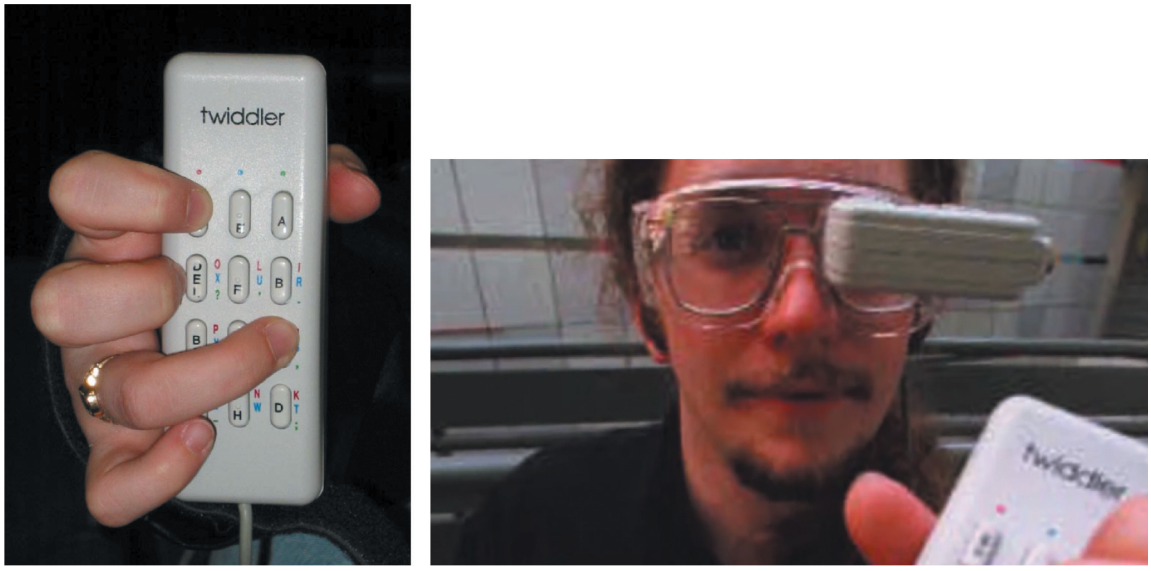
In measurements, T9's predictions were quite accurate so that it averaged 1.0072 KSPC, meaning the user only rarely needed to use the * key [MacKenzie and Soukoreff 2002b]. The measured speed was 15 wpm for novices and up to 40 wpm for experts [MacKenzie and Soukoreff 2002b].

8.3.4 Small Physical Keyboards

A key distinguishing factor for the RIM Blackberry devices (see Section 3.15 and Figure 3.23) was the small physical keyboard. These were typically operated with two thumbs. Although the idea of having a tiny keyboard was not novel, RIM was able to get patents on the way the keys were angled to make them easier to hit accurately with the thumbs (Figure 3.23) [Griffin et al. 1998]. One academic lab study measured the text entry speed of thumb keyboards that were a little larger than those on the RIM devices at about 30 wpm during the first session, which increased to an average of 60 wpm by the 20th session [Clarkson et al. 2005]. As mentioned above (Section 3.15), many PDAs adopted this kind of thumb keyboard in place of their gestural input techniques starting in the early 2000s, and thumb keyboards remained popular until replaced by the iPhone and its style of virtual keyboard in the early 2010s.

8.3.5 Portable Chording Keyboard

Another physical keyboard for mobile text entry is the Twiddler device, which is about the size of a TV remote control, and is designed to be used with one hand, without looking (see Figure 8.11(a)). These were introduced in the 1990s for use with wearable computers and is now (2022) available for about $200 from twiddler.tekgear.com/. The Twiddler has 16 keys arranged on multiple surfaces, so they can be operated with the fingers and thumb while holding the device. There is also a joystick on the top for moving the pointer cursor (in place of a mouse or touchpad). Chords of multiple buttons at once can input everything on a keyboard, plus customized strings like commonly typed names or email addresses. A research article [Lyons et al. 2004] reported that novices start off at 4.3 wpm (which is frustratingly low), reached 26 wpm after 400 minutes of practice, and then got to an average rate of 47 wpm after approximately 25 hours of practice in varying conditions. One subject achieved a rate of 67 wpm, matching that of Thad Starner, who has been a Twiddler user for many years (see Figure 8.11(b)).



(a) (b)

Figure 8.11 Versions of the Twiddler (a) used in the 2004 research study [Lyons et al. 2004] and (b) in use by Thad Starner with a heads-up display. Sources: (a) Fig. 1 of Lyons et al. [2004]. (b) Reused with permission, Boston Globe/MIT.

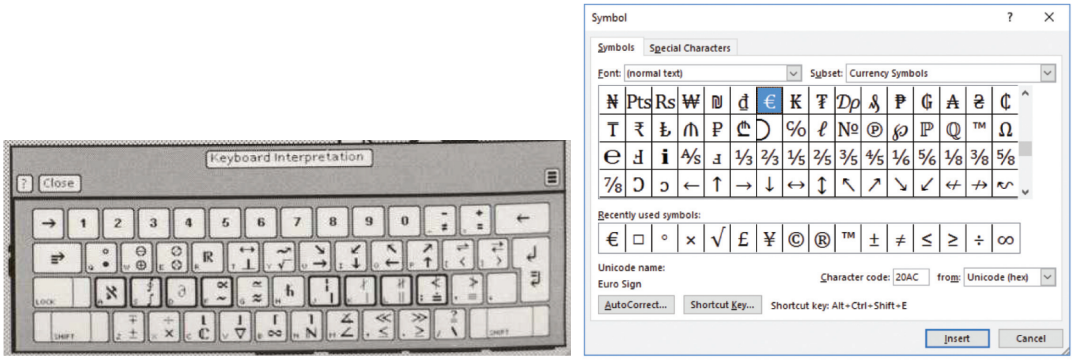
8.3.6 Virtual Keyboards

When a keyboard is displayed on a screen, it is called a *virtual keyboard*, *soft keyboard*, or *on-screen keyboard*. These keyboards are used for text entry across a wide variety of devices. The most obvious are the keyboards of smartphones and tablets, but there are many other uses.

For example, providing text entry to a cable-TV box, VCR, game console, or TV generally uses a virtual keyboard on the TV, and the user selects the desired key with a remote control. The keyboard may be in alphabetical order (see Figure 8.12) or QWERTY, and the control will typically use the arrow keys on the remote control (see Figure 4.4(b)), but on the Nintendo Wii system (see Figure 4.16(a)) the remote control wand is tracked so users can point at the keyboard on the TV from across the room. A related use for virtual keyboards is in virtual or augmented reality, where a keyboard may appear in the scene, and the user can point or select the desired keys.

Virtual keyboards on regular computers have long been used to enable the entry of special characters. For example, the Xerox Star from 1981 had an on-screen symbol keyboard that the user could click on with the mouse, or it remapped the regular keys to input those characters (see Figure 8.13(a)). Similarly,

Figure 8.12 Virtual keyboard displayed by the Xfinity (Comcast) set-top box for searching. The user moves the character selected using the remote's arrow keys. Source: Photo of screen by author on September 3, 2019.



(a) (b)

Figure 8.13 Virtual symbol keyboards (a) on Xerox Star from 1981 and (b) on Microsoft Office 2016. Sources: (a) Courtesy DigiBarn Computer Museum, http://www.digibarn.com/collections/screenshots/xerox-star-8010/xerox-star-8010-01.jpg. (b) Screen capture, used with permission by Microsoft.

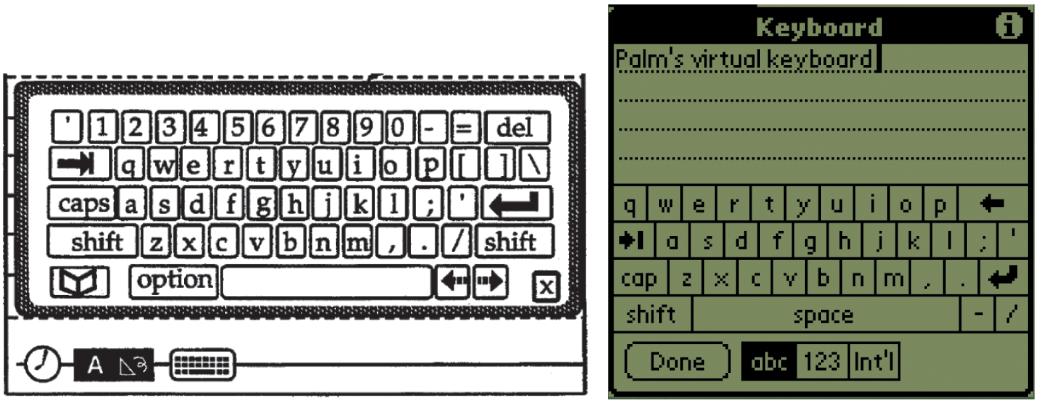
Microsoft Office displays a virtual keyboard when the user asks to Insert/Symbol (see Figure 8.13(b)), but it does not remap the regular keyboard, so it can only be operated by clicking with the mouse or moving the selection with the arrow keys.

Another application for virtual keyboards is to enable text entry on regular computers for people with physical disabilities. For example, for a person whose interaction is limited to a single button, a *scanning keyboard* is displayed on the screen,

with a highlight that moves typically from row to row, and upon pressing the button the highlight switches to move horizontally across the row until the desired key is under the highlight, at which point that key is entered. Text entry rates can be as low as 1 wpm for people with severe disabilities [Damper 1984].

Mobile devices have been the main use for virtual keyboards. Although handwriting was the featured way to enter text into the Apple Newton from 1993 (Section 3.8), it did provide an on-screen keyboard that could be popped up and tapped on with the stylus (see Figure 8.14(a)). The virtual keyboard on Palm devices was similar (Figure 8.14(b)) and was popped up by hitting the lower-left corner of the Graffiti input area (Figure 3.18).

Modern smartphones use similar virtual keyboards arranged in the familiar QWERTY layout. Many smartphones use predictive technology not only to provide autocomplete and auto-correct (covered below in Section 8.3.7) but also to help with keying accuracy. For example, smartphones increase the target area of expected keys. That is, if the system predicts that the next letter is more likely to be an "e" than a "w" or "r", then the area that the user can hit to get an "e" becomes bigger, even though the graphics on the screen remains the same size. Also, the iPhone provides the current letter in a popup above the key (Figure 8.15(a)), so it is visible. Some keys will pop up alternatives if the user presses and holds (Figure 8.15(b)). Because of these kinds of strategies, the iPhone was arguably the first small full virtual keyboard that could be used successfully with fingers (as opposed to with a stylus) when it was released in 2007.



(a) (b)

Figure 8.14 (a) The on-screen keyboard from the Apple Newton would pop up when the user tapped the keyboard icon at the bottom. (b) Palm's virtual keyboard for letters. Sources: (a) Fig. 8 of Capps et al. [1993]. (b) Screen capture.

(a) (b)

Figure 8.15 (a) The iPhone shows the current key in a popup since the key is hidden by the finger. (b) Some iPhone keys pop up alternatives that the user can select. Sources: (a) Fair use. (b) iOS screen capture.

8.3.6.1 Research on Virtual Keyboard Layouts

Like all PDAs of the time, the keyboards shown in Figure 3.18 had tiny targets because they were designed to be tapped with a stylus. Researchers realized that the default QWERTY layout was not very efficient for use with a stylus since the user had to move back and forth across the keyboard for many English words. This inspired research into alternative layouts that would be faster and more accurate for tapping with a single stylus (multi-finger typing on virtual keyboards was not expected since there is only one stylus). One example is the *OPTI keyboard* (Figure 8.16(a)), which was designed using a mathematical model that took into account the frequency of letter pairs in English to try to put closest together those letters that a user will need to type adjacently [MacKenzie and Zhang 1999]. For example, the letters in "the" are next to each other. The authors calculated the predicted upper-bound entry rate of 58.2 wpm, which is about 35% faster than the predicted rate for a QWERTY layout [MacKenzie and Zhang 1999]. There are four space bars to minimize its distance from all keys. Another example is the *Metropolis Keyboard* (Figure 8.16(b)), which recognized that the keys did not necessarily need to be rectangular and also tried to optimize the size of the keys [Zhai et al. 2000]. It used a different optimization algorithm but still tried to provide an increased

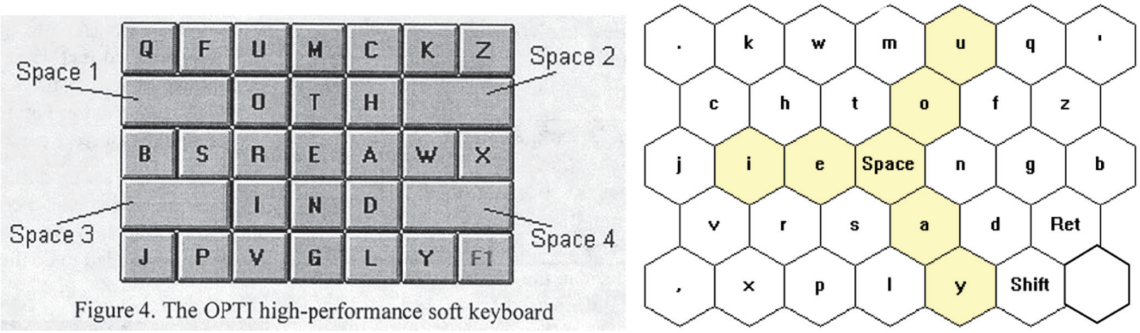


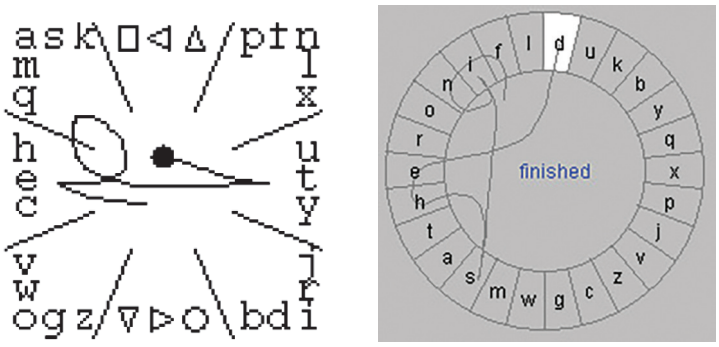
Figure 4. The OPTI high-performance soft keyboard

Figure 8.16 (a) The OPTI optimized virtual keyboard. (b) Metropolis virtual keyboard. Sources: (a) Fig. 4 of MacKenzie and Zhang [1999]. (b) Fig. 12 of Zhai et al. [2000].

speed of input based on stylus movement distances for English words—they calculated the top speed would be about 43 wpm [Zhai et al. 2000]. Other layouts have been explored by other researchers as well. However, none of these have caught on, despite their theoretically higher input rates, for four important reasons: (1) they are considered to be too hard to learn; (2) they have insufficient advantages to overcome the learning barrier; (3) they do not consider multiple finger input (e.g., two thumbs), which is now the norm on smartphones, and (4) they do not take into account the need for *backspace*, which is one of the most frequently used keys.

8.3.6.2 Gestures Over Virtual Keyboards

One of the bottlenecks with virtual keyboards is the requirement to move the finger or stylus down and up to tap on the desired key, in addition to moving around the keyboard to get into the right place. This has inspired a set of research and development around ways to enable text entry without needing to lift the finger or stylus off the screen. One early example was Quikwriting [Perlin 1998], where the characters were arranged in a square around the initial contact point, and the user could move from the center out to the desired character, then back or across (Figure 8.17(a)). Since the characters included space and enter, users could enter multiple words without ever lifting the stylus off the surface. A related research system was Cirrin [Mankoff and Abowd 1998], which put the characters around a circle, either in alphabetic order or using an order optimized to reduce the distance traveled for the most common words (see Figure 8.17(b)—note that the letters of "the" are contiguous). A character was entered by crossing into and out of its area in any direction. The first author reported that she got up to about 20 wpm after two months of use.



(a) (b)

Figure 8.17 (a) Entering the word "the" with Quikwriting: the user started at the dot, moved to "t" and back, then moved to the area for "h", which was at the top of that area so the user moved from that area upward, followed by re-entering that area for "e". (b) Entering the word "finished" with Cirrin required moving the pointer into and out of each desired letter. Sources: (a) Fig. 2 of Perlin [1998]. (b) Fig. 1 of Mankoff and Abowd [1998].

Researchers observed that with these gestural systems experts often learned the *shape* of the gestures for certain words. This is consistent with the findings for circular menus (Section 7.3.9.1), where drawing the memorized path through submenus can be done quickly. Shumin Zhai's group at IBM (see Chapter 35) applied this idea to performing gestures over an optimized virtual keyboard in a system called SHARK (which stands for Shorthand Aided Rapid Keyboarding) [Zhai and Kristensson 2003, Kristensson and Zhai 2004]. An important difference between SHARK and the previous systems was that the user could cross unwanted characters as the stylus moved from one desired character to another, which made some strokes ambiguous. The layout of the keys was optimized to minimize this based on English word frequency, but some ambiguity was unavoidable, especially for short words. This was further exacerbated by SHARK's recognition algorithm that relied primarily on the *shape* of the gesture, allowing users to be sloppy about actually hitting the desired keys for increased speed since users did not have to be as careful about the stylus placement. IBM spun out a startup called *ShapeWriter* (Figure 8.18(a)) that commercialized this concept on either an optimized or QWERTY virtual keyboard on smartphones and was released in 2008 [Zhai et al. 2009].

Meanwhile, another company developed the Swype input technique [Kushler and Marsden 2006], which was similar (Figure 8.18(b)). Both Swype and ShapeWriter were on the market simultaneously for a while (with Swype having the greater market share), and there were significant patent infringement legal battles between them. Eventually, the Nuance Communications company purchased the

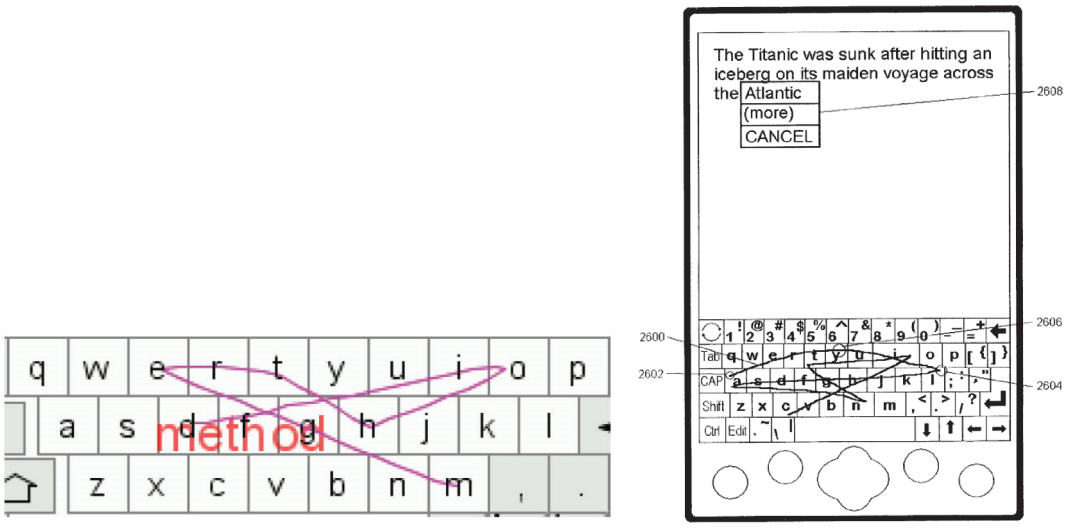
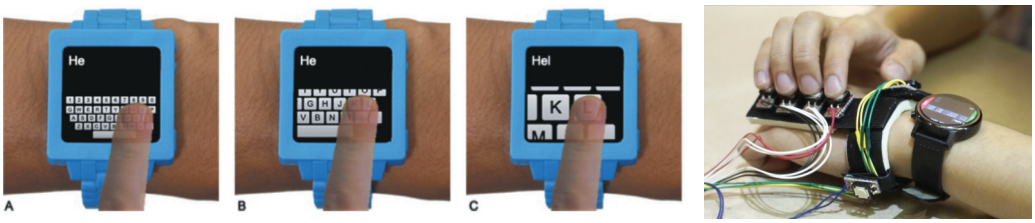


Figure 8.18 (a) ShapeWriter. (b) Swype. Sources: (a) Fig. 1 of Zhai et al. [2009]. (b) Fig. 2F of Kushler and Marsden [2006].

ShapeWriter company in 2010 and the Swype company in 2011. Nuance quickly killed ShapeWriter but kept the Swype version in the market for iOS and Android, but Nuance then discontinued support in 2018. Starting with iOS version 13.1.2 in 2019, a similar feature is now built-in to the iPhone virtual keyboard.

8.3.7 Text Entry on Wearables and Watches

Wearable devices that can be incorporated into clothing or worn on the body, like watches, necklaces, bracelets, rings, eyeglasses, and so on, often do not support text entry on the device at all due to the tiny size of the display and potential input area. Exceptions include the Fossil Palm Watch (see Figure 3.19) from 2003, which supported Palm's Graffiti text entry gestures (Section 8.3.2), and the Apple Watch, which has supported hand printing letters with a finger on the watch face (called Scribble) since Apple watchOS v.3.0 in 2017 [Apple Inc. 2017, Reich 2017]. Other methods already noted include using a separate device such as the Twiddler (Section 8.3.5), which is worn on a hand to provide input to a display worn as eyeglasses, and just using a smartphone's regular text entry methods to provide input to a wearable device communicating with the smartphone through Bluetooth or other wired or wireless means. Another option is to display a virtual keyboard somewhere (such as on an eyeglasses display) and to point to the desired character through a variety of pointer means, such as those discussed in Section 4.11. Newer



(a) (b)

Figure 8.19 (a) ZoomBoard zooms into the part of the keyboard where the user taps, so the keys get to be accurately tappable. (b) Watchboard prototype text entry using physical keys that might slide out from a watchband. Sources: (a) Excerpt from Fig. 1 of Oney et al. [2013]. (b) Excerpt from Fig. 1 of Lockerman et al. [2017].

devices, such as the Android or Apple watches, support speech dictation for text entry, where the speech recognition is generally handled by an off-device server connected through a wireless network.

None of the text entry methods for mobile devices (and especially for wearables) achieves the speed and accuracy that people can get with a conventional keyboard (see Section 8.5) or even the speed at which people can handwrite (see Table 8.1). Therefore, research and new products continue to address this situation.

For example, ZoomBoard (see Figure 8.19(a)) displayed a tiny QWERTY keyboard that zoomed into the part on which the user taps, so with two or three taps, each one zooming in further, the keys were big enough so that the user could hit the desired key accurately [Oney et al. 2013]. The authors reported that after eight practice trials, users achieved an average of 9.3 wpm. WatchWriter combined a language model to disambiguate taps or Swype-like gestures on a tiny QWERTY keyboard [Gordon et al. 2016]. Despite its small size, participants achieved an average of 22 wpm when tapping and 24 wpm with gestures. One group of students in my IxT class from 2016 tried creating a physical set of five to seven keys that could slide out from behind a watch with chords on the keys as a text entry method that they called *Watchboard* (see Figure 8.19(b)), but it only achieved an average of about 15 wpm after lots of practice [Lockerman et al. 2017].

A commercial example of a novel text entry method was the device called Tap from Tap Systems, Inc. [Tap Systems Inc. 2019], which used accelerometers on each finger to detect tapping of chords that represented each character to be entered (see Figure 8.20). The device communicated with any other device using Bluetooth and could be used to move a pointer by moving the thumb along a surface. The company's advertising reported that "the average tapper who completes 20 days of TapAcademy can tap at 35 wpm" [Tap Systems Inc. 2019].

Figure 8.20 Accelerometers on each finger of the Tap Systems device allow text entry by tapping chords on any surface. Moving the thumb along a surface moves the pointer like a mouse. Source: www.tapwithus.com [Tap Systems Inc. 2019].

8.4 Auto-Fill, Auto-Correction, and Autocomplete

An important component of text entry to computers is the various ways that the computer can help the user enter characters more *quickly and correctly*. Three important aspects of this are *auto-correction*, *autocompletion*, and *auto-fill*, which are different but often use similar or the same interaction techniques and implementation algorithms, so they are all covered in this section. These are extremely popular features and are relied on for almost all text entry to regular and mobile computers.

Auto-fill (also "autofill") is when the computer fills in a text field (see Section 8.7) *before* the user has entered anything into that field. A modern example is when a web browser detects an address field on a webpage and automatically fills in the user's address before the user has entered any text into that field.

Auto-correction is when the computer *changes* the characters that the user has already input with the intent of correcting the spelling, grammar, or other aspects of the entered text (including the capitalization, punctuation, etc.). For example, if the user types "teh", the computer might change it to "the". Note that the computer is *trying* to correct the entered text—sometimes auto-correct actually changes the spelling to be *incorrect*, especially for unusual or foreign words, or proper nouns.

In contrast, *autocompletion* is when the computer adds in *new* characters to what the user has already entered, with the intent of finishing the word or phrase for the user. This definition also recognizes that the autocompletions will sometimes be incorrect. For example, if the user types "compl", the computer might fill in the rest of "completion". Autocomplete is also called *auto-prediction* (since it is *predicting* what the user will type), *word completion*, or *word prediction*.

It is possible for both auto-correct and autocomplete to be performed at the same time. For example, if the user types "comzpl", the computer might correct and complete it to be "completion" as well. The interaction techniques for all of these are covered next.

8.4.1 Auto-Fill

Auto-fill is differentiated from the other ways that text is automatically added in that it happens *before* the user enters anything. For example, for the form fill-in interaction style (Section 2.5.1.1) where users enter text into fields of the form, in many early systems (dating back to the 1970s), if there was a default value, the system would fill that in automatically [Smith and Mosier 1986]. As discussed below (Section 8.6.3.4), often the entered text will be in *pending delete* mode, which means that the default text is deleted if the user moves a cursor into the field and/or types any characters.

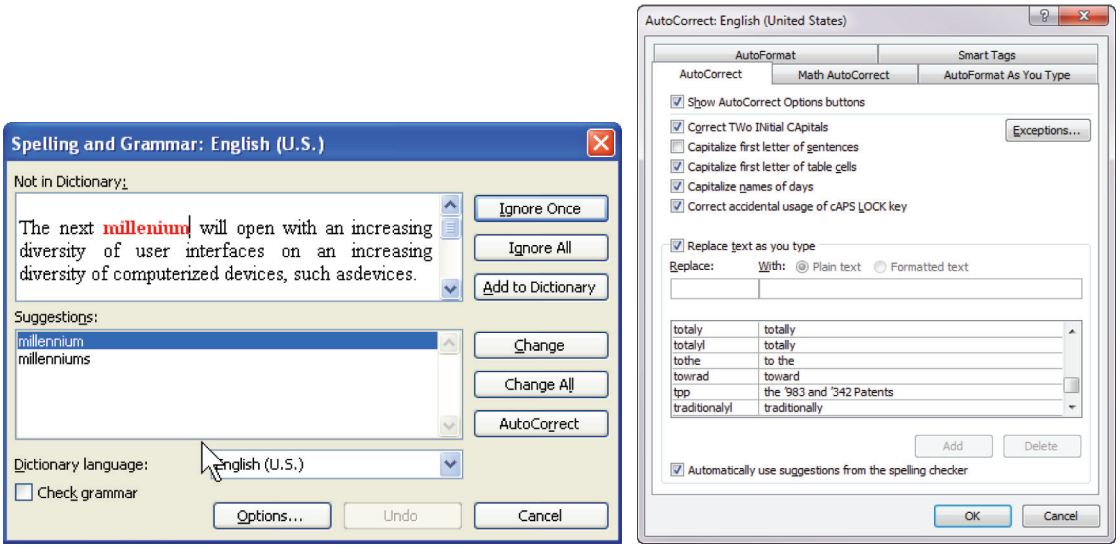
While auto-fill has always been part of form fill-in, it has gained more prominence with features of web browsers, which auto-fill constant user information such as the user id and password. Typically, browsers have a table that maps specific web pages to the page-specific information, such as when users have different passwords for different websites, plus other cross-site information that will be filled in automatically like a user's home address and phone number.

8.4.2 Spell Checking and Auto-Correction on Regular Computers

A famous early system for correcting text was *Do What I Mean* or DWIM, described by Warren Teitelman in his 1966 MIT Ph.D. dissertation [Teitelman 1966]. DWIM used a variety of heuristics to automatically fix typos and other common typing mistakes, such as typing a neighboring key on the keyboard.

Probably the earliest widely used forms of automatic correction of text were *spell checkers*, which also date back to the 1960s (see Kukich [1992] for a survey of early systems). These typically operated in batch mode on a file of text using a dictionary and algorithms for handling *morphology* (which is how words are changed to be plural, past-tense, possessive, etc.). Spell checking was available as separate applications and was built into text editors such as WordStar and WordPerfect from the 1980s. Spell checking appeared in Microsoft Word in version 5 in 1991 as a modal process that interactively allowed users to see each candidate misspelled word and select a replacement, ignore this instance or all instances, or else to add the unknown word into the dictionary (see Figure 8.21(a)). Grammar checking was also available, checking for such problems as noun–verb agreement or using the wrong kind of "it's." In Word, the first implementations of some of these were not particularly accurate, inspiring many people to investigate how to turn them off, such as unchecking the "Check grammar" checkbox in Figure 8.21(a).

In addition to the spell checker, Word version 6.0 in 1993 introduced its "AutoCorrect" feature, which used a large, user-editable table listing what the user would type and what the system would change it into, such as "tothe" to "to the" shown in Figure 8.21(b). This mechanism continues to be used in Word today. All of these



(a) (b)

Figure 8.21 (a) Spell checker in Microsoft Word 2003. (b) Auto-correct in Microsoft Word. Sources: Screenshots, used with permission by Microsoft.

changes are put into the undo stack so the user can undo the automatic changes. For example, if the user types "teh<space>", the text will change to "the<space>", and hitting undo (^Z) will go back to "teh<space>", and another undo will delete the entered space (leaving "teh")—thus two items are added to the undo stack at the time when the user only entered the one <space> character (see also Chapter 15 for a full discussion of *undo*).

The next important advance in auto-correct was the *wavy underlines*, also called *squiggly underlines*, which were introduced in Word 95 from 1995 [Mogilevsky 1995] (see Figure 8.22). In addition to the interaction technique, this also requires checking the spelling incrementally in the background as the user types [Mogilevsky 1995], so it is also called *background spell checking*. Originally only used for spelling (red wavy underlines), new colors were introduced later (1997) for grammar errors (green then later blue, and then in Word 365, double-blue underlines that are not actually wavy anymore—see Figure 8.22(c)), formatting errors, and so on. Right-clicking on the underlined word pops up a menu of proposed fixes (see Figure 8.22).

Wavy underlines were adopted by code editors for programming languages (also called an *integrated development environment* or *IDE*), which requires that the compiler (which checks the code for errors) be able to run in the background.

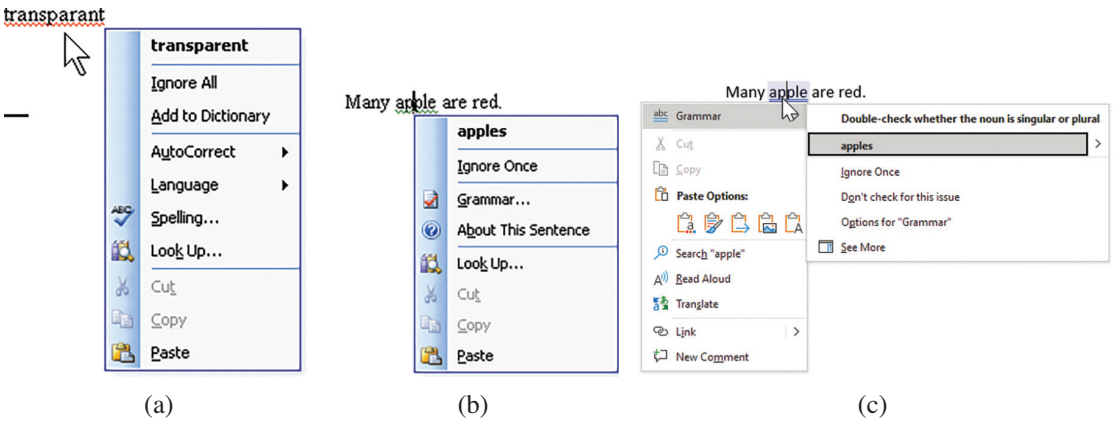


Figure 8.22 Wavy underlines: (a) red for spelling and (b) green for grammar. (c) Word 365 still uses red wavy underlines for spelling but switched to double blue underlines for grammar. Sources: (a) and (b) Screenshots of Microsoft Word 2003. (c) Screenshot of Word 365. Used with permission by Microsoft.

This feature first appeared in Visual Studio.NET 2002 IDE for Visual Basic.NET [Abraham 2005]. Similar features were adopted for most programming languages and by most other IDEs. Right-clicking on the underlined word, or on the associated icon in the margin, pops up a menu of possible corrections for the problem, as shown in Figure 8.23 from the Eclipse IDE. These are sometimes called *quick fixes*.

8.4.3 Autocomplete on Regular Computers

Autocomplete (the computer filling in more than the user types) is most useful for mobile devices (Section 8.4.4), but it started on regular computers and still has some uses there. For example, Word 365 and Google Docs now provide one

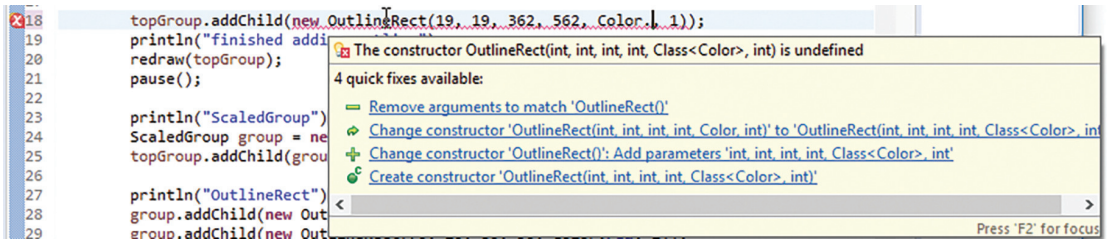


Figure 8.23 Wavy underline in the Eclipse IDE, with auto-correct menu of quick fixes. Clicking on the red "x" icon in the left margin also pops up the quick fixes menu. Also shown are the different colors for the code elements. Source: Screenshot of Eclipse Version Neon.3 Release (4.6.3) from 2017.

proposed completion that the user can accept by hitting space or tab or ignore by just continuing to type (Google calls this *Smart Compose*, and it was first released in 2020 and uses machine learning algorithms). Like auto-corrections, completions date back at least to the 1960s, starting from systems that helped users enter computer code. Autocomplete continues to be an important feature of code editors, where programmers typing their code rely on the system to pop up a list of choices. For example, Microsoft's code editors (or IDEs) have had a feature called *IntelliSense* dating back to around 1996 that pops up a list of completions after a user types a period (see Figure 8.24). This feature, also called *code completion*, quickly appeared in other code editors and is ubiquitously expected today. In my group's research, we have expanded the ways that code completion can help programmers, such as suggesting other code that users may want to insert [Mooty et al. 2010, Santos and Myers 2017], and popping up custom GUI tools that generate code as users interactively select the desired item, such as a desired color [Omar et al. 2012].

Similar approaches were studied for entering other long strings, such as having the computer fill in the full names of journals by typing a few letters [Jakobsson 1986]. The first version of the Reactive Keyboard predicted what would be typed in a Unix command line interface to help with entering commands, arguments, and filenames (1988), and the next version helped with entering free text on the Apple Macintosh (1990) [Darragh et al. 1990]. It used an adaptive algorithm that learned from what the user previously typed. The Microsoft Word auto-correct feature mentioned above (Figure 8.21(b)) can also be used for autocomplete by adding a shortcut into the table. For example, a user can enter "soit" into the left column and "The Story of Interaction Techniques" into the right column, and then typing "soit<space>" is replaced with the full title. However, as pointed out

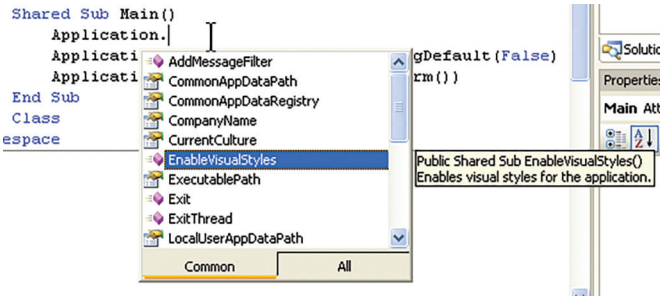


Figure 8.24 Visual Studio pops up a menu of autocompletions after the user types a ".". Source: Screenshot of Microsoft Visual Studio 2005. Used with permission by Microsoft.

by Darragh et al. [1990], most words in English are fairly short, and for reasonably skilled typists and regular text entry it is faster to type the full word on the keyboard than to have to pop up a list of options and pick one. On the other hand, for people with disabilities who use input mechanisms like scanning keyboards (Section 8.3.6) or head-mounted pointing devices, autocomplete on regular computers can be significantly helpful [Darragh et al. 1990]. Further, the new machine learning-based autocomplete systems like Google's Smart Compose or even GitHub Copilot from 2021 and Generative Pre-trained Transformer 3 or 4 (GPT-3 or GPT-4) can enter whole phrases, or even large blocks of code, with a simple interaction technique for confirming, making it more attractive for general text entry.

One important recent use for autocomplete is to help users enter information into web forms. One example of this is that web pages may pop up a list of cities after a user has entered a few letters into a destination or address field (Figure 8.25). Similarly, a web browser may pop up menus containing user-specific information such as the user's phone numbers or credit card numbers.

A more interesting recent use for autocomplete (combined with auto-correct) is the way that search engines such as Google provide options as the user types into the search field (see Figure 8.26). Google's version of this is called Google Suggest, dating from approximately 2004 [Gannes 2013], and it has also been available

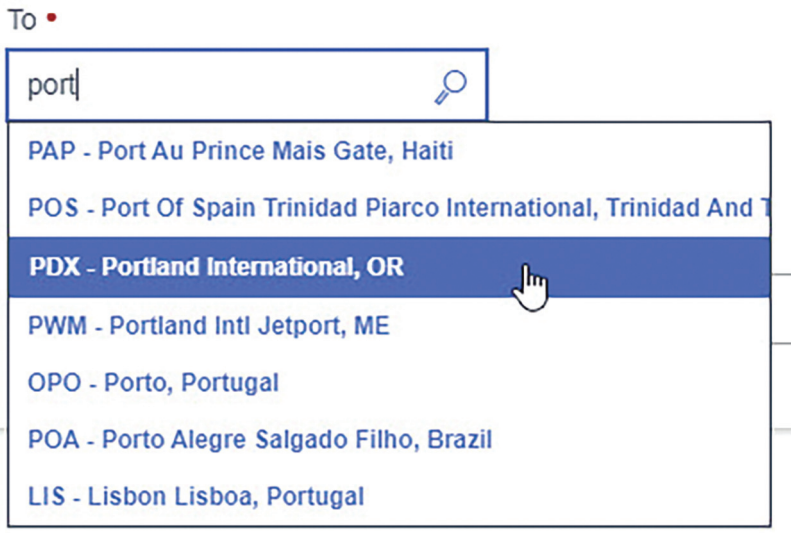


Figure 8.25 The AA.com website provides a list of autocomplete options in the destination field. Source: Screenshot of AA.com website on January 15, 2021.

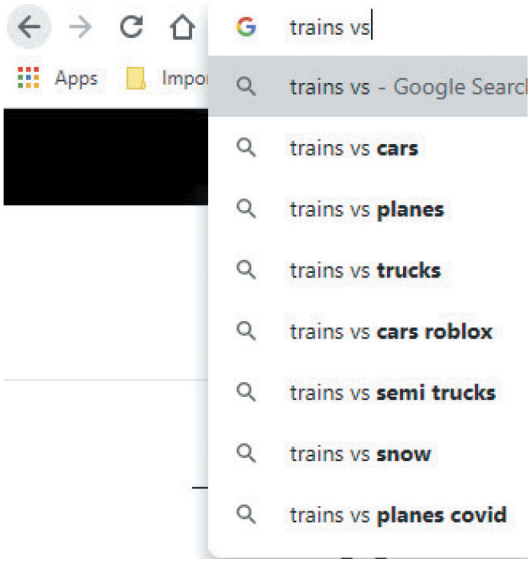
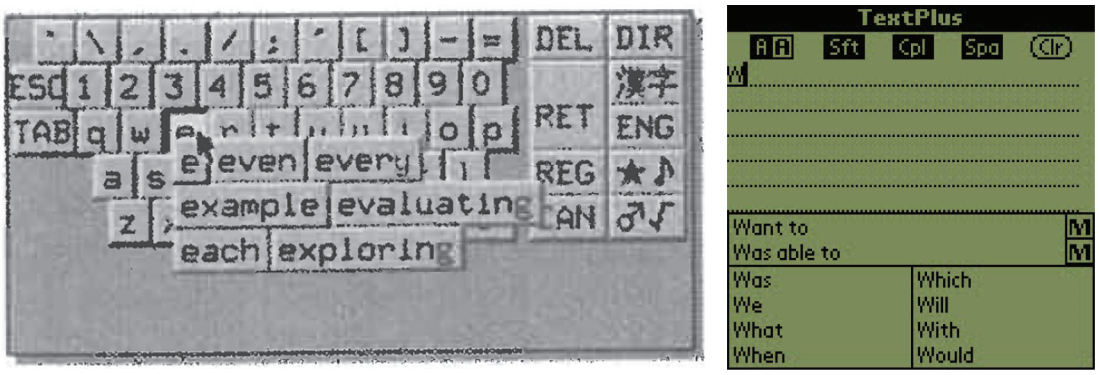


Figure 8.26 Chrome search bar (address bar) autocomplete. Source: Screenshot of Google Chrome version 87.0.4280.141 in 2021.

from other search engines such as Yahoo [Sullivan 2008] and is now ubiquitous in search boxes for many websites. This feature, which is also called *autosuggest* or *incremental search*, takes advantage of both the users' individual contexts (such as their location and history of searches) and the current global context (such as what topics are currently newsworthy).

8.4.4 Auto-Correction and Autocompletion on Mobile Devices

Given the smaller, slower, and more error-prone input methods on mobile devices, it is not surprising that auto-corrections and autocompletion are more important and more universally used on them. One early example was the POBox research system from 1998 for stylus-based PDAs, which used a fixed dictionary of words. As the user typed letters, the system provided the most popular word with the typed letters (see Figure 8.27(a)). After the first word, the system would predict the next word even before any letters were typed. The measured input rate in a lab study was only about 8 wpm [Masui 1998]. Companies created versions of autocomplete for Graffiti such as TextPlus by SmartCell Technology (see Figure 8.27(b)), which was released around 2000 [MacKenzie et al. 2006]. Research systems continued to explore other autocompletion and correction mechanisms such as Unipad, which focused on movement minimization and reduced attention demand [MacKenzie et al. 2006], and the EdgeWrite in-stroke word completion described in Section 8.3.2.



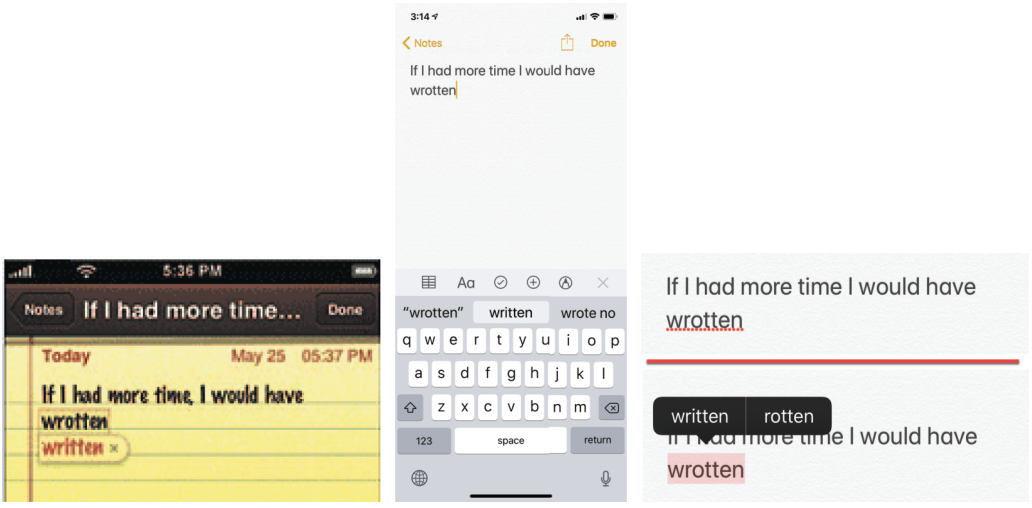
(a) (b)

Figure 8.27 (a) POBox autocomplete—when the stylus touches the "e" key, it pops up likely words and the user can slide to select or release to pick only the letter. (b) TextPlus autocomplete for Palm from SmartCell Technology in 2000. Sources: (a) Fig. 8 of Masui [1998]. (b) Archive of SmartCell website from June 9, 2000: https://web.archive.org/web/20000609160120/http://www.smartcell.com/textplus.html.

In 2007, the iPhone was unveiled, with a number of radical departures in its design for auto-correct and autocomplete. As described above (Section 8.3.6), the iPhone's virtual keyboard was one of the first designed to be used with fingers, and Apple wanted to facilitate high-speed and reasonably accurate text entry—so autocomplete and auto-correct were essential. A key innovation in the original iPhone design was that the autocomplete or correction was used *by default*, and the user had to perform an action to *cancel* its use. As discussed above, all previous systems popped up a list of options from which the user had to pick in order to use a recommendation. The original iPhone displayed a single option in a popup box (Figure 8.28(a)). If the user hit a space, return, or punctuation, the recommendation would be used. To cancel using it, the user had to tap on the box itself, which proved confusing (in spite of the little "x" in the popup to signal deleting it) since you would point to the correction to show you did *not* want it.

This approach was adopted by other smartphones. Android smartphones supported third-party keyboards (which the iPhone would not get until 2014), and in 2010, a commercial product called SwiftKey for Android provided three predicted words (see Figure 8.29) that would be used as autocomplete or to predict the next word after a space [Hamelin 2010].

Starting with iPhone's operating system version iOS 8 in 2014, Apple changed the autocomplete to use a system similar to SwiftKey's of showing multiple options above the keyboard (Figure 8.28(b)), which the user can tap on to select. When it is confident about the prediction, one will be shown with a white background, which



(a) (b) (c)

Figure 8.28 (a) Original iPhone's auto-correct and complete suggestions were shown in a popup and used by default. (b) Since iOS 8, three suggestions are shown above the keyboard (this picture is from iOS v. 12.4.1). (c) Misspellings have been marked with a dotted underline, which when clicked display corrections. Sources: Screen captures of original and current iOS features.

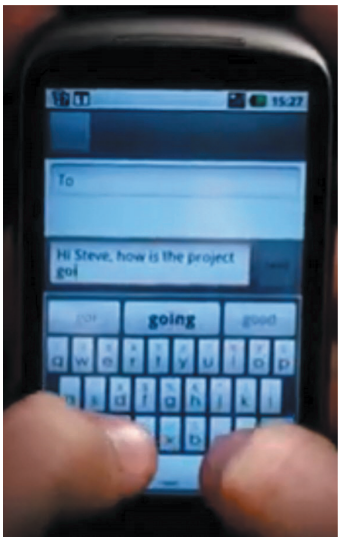


Figure 8.29 SwiftKey autocomplete product from 2010 displays three words on Android. Source: Hamelin [2010].

will be used by default if the user types a space or equivalent. If a misspelled word is left in the text, it will have a red dotted underline, and tapping on it will pop up corrections that can be selected by tapping on the desired one (Figure 8.28(c)). Apple calls these features QuickType.

All of Apple's predictions use multiple previous words to compute the most likely words to propose. This means that the most probable completion depends on what was already entered. For example, if I type "Brad m", the iPhone will propose "Myers" for the "m", but if I type "Who m", the iPhone proposes "might" or "may". This also illustrates that the iPhone predictions are adaptive, and it learns over time what a user is most likely to type.

Since the iPhone's corrections are used by default, people are more prone to not noticing that words are mistakenly changed, possibly to the wrong word. This has led to many funny and embarrassing messages being sent, such as those documented in the former website "damnyouautocorrect.com", such as shown in Figure 8.30. I conjecture that Apple made this decision because the intended use for text entry on the iPhone was for text messages and other informal situations, in contrast to writing formal letters or papers on a regular computer, so having incorrect words in the text would not be as significant. Moreover, the added speed of not needing to look at and check the corrections would make autocorrect by default worth it. Still, there have been requests on Stack Overflow and other question-and-answer sites about how to turn off auto-correct, which is possible for the programmers who create an app or a web page (so they can disable autocomplete and auto-correct for particular text entry fields), and for the end-user who can turn them off altogether with a global setting. Autocomplete is also available

Figure 8.30 Example of auto-correct changing the content in unexpected ways. Source: Archive of damnyouautocorrect.com from Feb. 10, 2011 (of a webpage from October 2010). https://web.archive.org/web/20110210015523/http://damnyouautocorrect.com/2277/the-10-most-popular-dyacs-from-october-2010/.

Figure 8.31 ListPad uses web services to provide context-dependent autocompletes, such as the sessions' names for a conference (at 3). Source: Chang et al. [2013].

for text entry for watches—for example, the Apple Watch Scribble text entry allows selecting the desired completion using the watch's knob [Apple Inc. 2017, Reich 2017].

Research continues on autocomplete techniques for mobile devices. For example, my group explored "ListPad" to provide context-sensitive autocomplete for structured lists, when items in the list come from a data source [Chang et al. 2013]. Suppose the user is typing a list of conference sessions to attend. By attaching ListPad to a web service that provides the session data, the user can start typing the name of a session and ListPad will match the user's typing to the session names to provide a row of autocompletes above the usual Android word list (see Figure 8.31).

8.5 Measuring Text Entry Speed and Accuracy

8.5.1 Speed

Throughout this chapter, the wpm for various input methods has been listed, but what is the right way to measure that speed? In particular, how does the wpm interact with the errors that users make, and their error corrections? For example, if the user types "Tezt<backspace><backspace>xt", then the user has typed eight characters (counting the backspaces and the final "t" was typed twice) but only four remain in the text. And typing those corrections took the user some amount of time. Further, what about errors left in the text, and automatic corrections and completions? Researchers, including in my group, have studied this issue [Soukoreff and MacKenzie 2001, MacKenzie 2002, MacKenzie and Soukoreff 2002b, Wobbrock and Myers 2006], and there are now well-accepted ways to measure text entry speed and accuracy [Wobbrock and Myers 2006].

The first question is: what is a "word" in "words per minute" since words can have a widely varying number of characters in them, and with most text entry methods, users actually enter individual characters. And how do spaces and punctuation count? Researchers have decided to simply define a "word" in this measure as five characters, including all spaces and punctuation, so wpm = characters per minute (cpm) divided by 5 [MacKenzie and Soukoreff 2002b]. Another measure is characters per second (cps), but for many input techniques that number will be a fraction (8 wpm = 40 cpm = 0.67 cps). Section 8.3.3.1 discussed KSPC as a measure of how many keystrokes on average a user needs to make to enter a character [MacKenzie 2002].

The simplest way to measure text entry speeds is to display specific words (possibly with punctuation) and require the user to enter exactly the displayed characters. If the user enters an incorrect character, then the system will not let the user continue to the next word until all the errors are corrected. Then counting the number of entered words over a period of minutes is a simple measure of the wpm. Since inaccurate entry methods will cause the user to need to spend time correcting the errors, the wpm indirectly takes into account the error-proneness of the entry method. A related method shows users erroneous words but does not necessarily force them to be corrected. The website 10fastfingers.com/typing-test/ uses this approach. In this case, the results are wpm plus a count of the number of words done correctly or incorrectly.

However, these kinds of tests are quite unnatural since they focus on word-by-word text entry instead of realistic sentences or phrases. In real life, people may notice errors in previous words and go back and correct them, and mistypings are often not noticed or highlighted. Therefore, more advanced analysis methods have been devised to provide more sophisticated reports of speed and accuracy, allowing people to enter realistic full phrases [Wobbrock and Myers 2006].

8.5.2 Learning

Another significant issue is *learning time*—how long does it take a user to learn a text entry method. Sometimes studies will show the *learning curve*, which shows how people's speed changes over time. Typically, this will have a logarithmic shape—people start slowly but get faster in the first few trials, and eventually the speed up dissipates and people reach their asymptotic speed (or equivalently, the *time*, which is the inverse of speed, goes down—see Figure 8.32). However, the curve rarely flattens out completely—people may continue to improve even with their own long-used devices. When reporting the speed and accuracy of a text entry method, it is important to be clear about which part of the learning curve was tested and how much training and practice the users received.

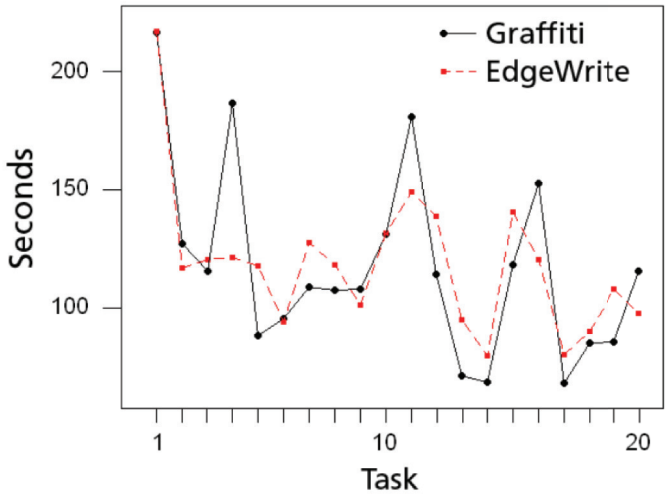


Figure 10. Average task times across all 20 tasks show that EdgeWrite and Graffiti were learned at comparable rates. The upward spikes in the downward trend are due to tasks with higher amounts of punctuation interspersed among simpler tasks.

Figure 8.32 The learning curve for the EdgeWrite text entry method (Section 8.3.2) goes *down* since it measures time. Source: Fig. 10 of Wobbrock et al. [2003].

8.5.3 Accuracy

To go beyond just word-level accuracy, researchers have provided ways to measure character-level accuracy [MacKenzie and Soukoreff 2002a]. For example, my group devised a method [Wobbrock and Myers 2006] that provides information about the speed and character-level errors in input streams for unconstrained text entry evaluations. An important observation from the new work is that the user may try to enter a character and get no response—for example, when no input at all is produced by an unrecognized gesture in Graffiti or by a tap on a virtual keyboard that is not detected by the sensor. Another observation that is rarely handled is that the user's corrections can themselves be in error—for example, the user might hit backspace by accident when trying to hit other keys or hit other keys when trying to hit the backspace. Our algorithm reports many kinds of errors (see Wobbrock and Myers [2006] for full details):

- *Uncorrected errors*: Errors that are left in the final string. These include:
  - *Uncorrected substitutions*: When the user typed the wrong character and left it in the final string, like "quicxly".
  - *Uncorrected insertions*: An extra character was entered, like "quickxly".

    - *Uncorrected omissions*: The user skipped entering a character, like "quicly".

- *Uncorrected no-errors*: The characters the user typed that are correct and have been left in the final string.
- *Corrected errors*: These are errors that the user noticed and corrected, and include:
    - *Corrected no-errors* are correctly entered characters that the user nevertheless erased (and often retyped later). This is typically done because the user notices an error earlier, such as after typing "quicklxy" the user might decide it is easier to type backspace twice, so the correct "y" is deleted and then retyped.
    - *Corrected substitutions* are when the user corrects a mistyped character and corrects it, like "qv<w<uickly" (where "<" means backspace) shows two corrected substitutions as the user is having a difficult time entering a "u".
    - *Non-recognition substitutions* are when the user tries to enter a character but nothing is recognized, but then the user goes ahead and enters the correct character afterward.
    - *Corrected insertions* are detected when the user enters an extra character and later goes back and deletes it, like "qxui<<<uickly". Note that the algorithm can only detect this as an *insertion* if there are extra corrected no-errors after it ("ui" here), because otherwise it would be classified as a substitution.
    - *Corrected omissions*, like insertions, require corrected no-errors so the algorithm can see that a character was actually missed, like "quikl<<ckly".

The designer of a new text input method might be interested in this level of detail about errors in order to reduce all the kinds of errors that users make and to enable swift error correction when they do happen. Two useful outputs of this analysis are the *character-level error rates*, showing how often each character is mistyped, and a *confusion matrix*, showing how often one character is entered when a different one is intended. The latter can be especially useful for optimizing a gesture set to better differentiate gesturally similar characters, or for virtual keyboards for determining the required separation or layout of keys. Note that this may result from a recognizer problem but alternatively may just be that the user is confused about how to enter the character [Wobbrock and Myers 2006].

8.5.4 Confounds from Auto-Correct and Autocomplete

A confound for measuring text entry speed and accuracy is autocomplete and auto-correct, for a number of reasons. First, the accuracy measures just mentioned cannot be used if the system is correcting the user's typing—presumably, the key accuracy measure will be uncorrected errors at the word-level since few character-level errors will be left in the text. Second, one might want to log the instances where the system did autocomplete or auto-correct in order to measure the effectiveness of those automatic features. Third, since these are usually *adaptive*, their behavior will change over time, so different users, and even the same users at different times, will see different corrections and completions. This can be especially problematic if the same phrases are used repeatedly. In a lab setting, the tester may be able to reset the automatic features (or not, depending on the system), but this is not appropriate if testing on the user's own device.

8.5.5 Target Text for Testing

Another consideration when designing or using a test for a text entry method is the target text that the user must enter. Ideally, the test should use text similar to what a user would enter using the text entry method. For example, on a mobile device, the text used for the test might come from a database of SMS messages, whereas text entry for a laptop might be measured on standard English as might be used in documents and reports. This can make a big difference since SMS messages might use abbreviations and much shorter words, whereas standard documents might use long sentences and more punctuation, which can be harder to enter using a virtual keyboard.

The phrase "The quick brown fox jumped over the lazy dog" is popular for text entry studies since it uses every English letter, but there are problems with just using one phrase over and over—the user will memorize the phrase and remember how to type it, which may not be realistic. Further, the autocomplete might start predicting the next word or the entire phrase, so the user only would need to tap a space character a few times, which is not a reasonable test.

The test 10fastfingers.com/typing-test/ uses random words that the user must type, whereas most researchers use realistic phrases or whole sentences taken from a corpus. The user could then look at the sentence and presumably remember multiple words, so the results are less confounded by the user's time for reading and looking at the target phrases [Kristensson and Vertanen 2012]. Real sentences are also more likely to be a good test of the effect of autocompletion since that predicts the next word based on previous words that the user has already entered. MacKenzie and Soukoreff [2003] published a phrase set in 2003 of 500 phrases that they claim are "moderate in length, easy to remember, and representative

of the target language" but which have "no punctuation symbols, and just a few instances of uppercase characters." These have been widely used for text entry studies [Wobbrock and Myers 2006], but these phrases are obviously problematic if one wants to test the effect of capitalization or punctuation, or the kinds of text most likely to be entered onto a smartphone, not to mention testing any language other than English. Therefore, many newer studies either use modified and translated versions of this phrase set [Ruan et al. 2018] or newer ones [Kristensson and Vertanen 2012].

Another issue is the instructions given to the users about the phrases and the test in general. We usually say: "enter the text as quickly and accurately as you can," but it is possible to bias users toward either higher speeds or higher accuracy by the instructions. One study tested the effect of requiring users to *memorize* the phrase before starting to type and found that this increased the speed but decreased the accuracy [Kristensson and Vertanen 2012].

8.5.6 Summary of Speed and Accuracies of Popular Methods

Given all the caveats and confounds mentioned above, it can be quite hard to compare text entry speeds of different methods. It can be especially problematic to take the numbers from different published accounts and try to compare them since they may have used significantly different methods to measure the speed and accuracy, different points on the participants' learning curve, different instructions to participants, and different kinds of target text. Some publications present a *theoretical* speed based on a model rather than an experimentally measured speed, and other publications are more like advertisements and show the speed of one of the best performers.

However, it is still somewhat useful to see the speeds reported for different methods, as shown in Table 8.1.

8.6 Text Editing and Formatting

The previous sections have been about text *entry*, or how users can get text into a computerized device. We now switch to *text editing*, which is how users can *change* the text after it has already been entered. This includes moving the text cursor, selecting text, deleting it, changing the text, and so on. In contrast, *text formatting* (sometimes called *typesetting*) is how users can specify and change the way that the text *looks*, which includes the text font, size, styling, layout on the page, margins, whitespace, and so on. Modern editors like Microsoft Word combine both editing and formatting features, whereas other text editors like Microsoft Notepad only operate on *plain text*, which is text without any formatting. (Note that Wikipedia currently (February 2022) defines a "text editor" as "a type of computer program

that edits plain text", but I disagree—I believe that it is appropriate to call Microsoft Word a text editor even though it edits formatted text.) Text *with* formatting is called *formatted text*, *rich text*, or *styled text*. Some current ways to do text formatting that are *separate* from a text editor are HTML or LaTeX [Lamport 1985], which involve adding into the plain text special commands, called *markup*, that describe how the text should look. These commands form a *text formatting language*, also called a *batch document processing system*. As a simple example, here is how to get "Text which is *italic* or **bold**" in two formatting languages:

- HTML: `Text which is <em>italic</em> or <strong>bold</strong>`
- LaTeX: `Text which is \textit{italic} or \textbf{bold}`

With these, the user takes a whole file and processes it with a program to produce the formatted pages typically in an encoding (like PostScript or PDF for LaTeX) that can be printed onto paper and, at least recently, can also be viewed on the screen.

This section surveys some important historical text editors and formatters and then discusses interaction techniques for text selection and editing.

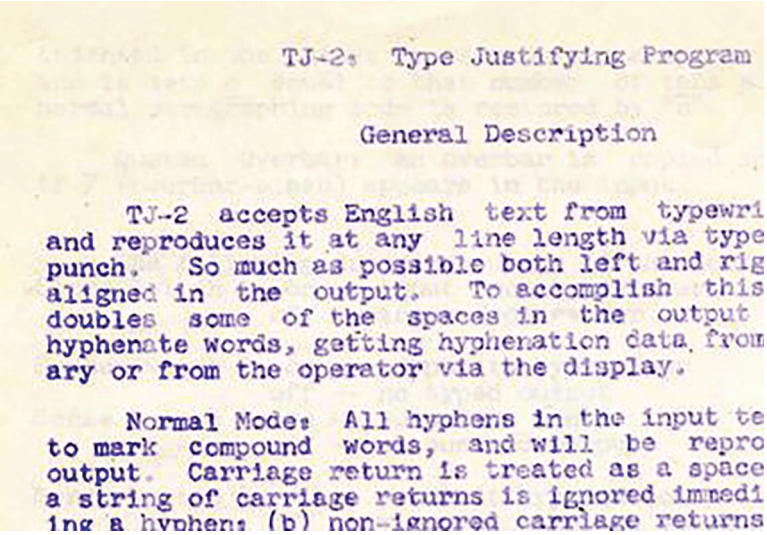
8.6.1 Brief History of Text Editing and Formatting

As soon as people were able to get interactive access to computers, which started in the 1960s with timesharing (Section 2.2.1), text editing was supported. When people only had a printing terminal (like Figure 2.3), it was very difficult to edit the file. Typically, people had to use what were called *line editors*, where one could edit the file line-by-line. Early examples include QED from 1965 (for "quick editor") and "ed" for the first version of Unix in 1969 [Salus 2005]. As an example, a user might type "i" (which means *insert*) and then type some content. Typing a period on a line by itself would cease the input. While typing, a user could only use the backspace key to fix what was input on the current line. After leaving insert mode, various editing commands were available that would typically operate on the current line, which was not visible, so the user had to remember it. For example, the user could delete the current line or substitute some text in the current line for new text, like: "s/better/bad" would change (**s**ubstitute) the current line from "This is better content" to "This is bad content". Users could also print out the current line or any desired section of the document on the terminal.

This was obviously very difficult and awkward, and a key advantage of display terminals (like Figure 2.7) was making text editing *much* easier since the user could see the contents. One of the impressive aspects included in the "mother of all demos" of NLS by Doug Engelbart in 1968 was editing of text on the screen using the mouse (Section 2.4.2), but most display terminals, which started around 1969,

only had a text cursor that could be moved using arrow keys or other keyboard commands. A variety of competing screen-based plain text editors were developed in the 1970s for these terminals, including vi, by Bill Joy in the Unix group of Bell Labs from around 1976 [Joyce 1984], and Emacs, also from about 1976 by Richard Stallman, Guy Steele, and Dave Moon the MIT AI Lab [Zawinski 1999]. Two key differences in the operation of these two tools are that vi is moded whereas Emacs is modeless (see Section 8.6.3.1), and Emacs has always been programmable so users can create their own commands. The descendants of vi (e.g., vim) and Emacs (e.g., GNUEmacs) continue to be used today, mostly by programmers. Most operating systems came with a simple plain text editor, like "Notepad" starting in 1983 for MS-DOS and then Windows.

Even though these editors only handled plain text, there was still a need for computer-controlled formatting. This inspired the creation of *text formatting languages* as defined above. The original programs, like TJ-2 from 1963 and RUNOFF from 1964, were designed for computer-controlled typewriters (like Figure 2.3) or other single-font, fixed-width line printers. They would fill paragraphs, hyphenate words, and add spaces to justify the right margins (see Figure 8.33). The only other formatting available at the time was the use of ALL CAPITALS or underlining, which



TJ-2: Type Justifying Program

General Description

TJ-2 accepts English text from typewri
and reproduces it at any line length via type
punch. So much as possible both left and rig
aligned in the output. To accomplish this
doubles some of the spaces in the output
hyphenate words, getting hyphenation data from
ary or from the operator via the display.

Normal Mode: All hyphens in the input te
to mark compound words, and will be repro
output. Carriage return is treated as a space
a string of carriage returns is ignored immedi
ing a hyphen; (b) non-ignored carriage returns

Figure 8.33 Formatting for a typewriter-like printing from TJ-2 in 1963. Source: Type Justifying Program (TJ-2) by Peter Samson, published in a PDP-1 Massachusetts Institute of Technology Memo PDP-9-1 dated May 9, 1963. Copyright holder for the image is the authors of TJ-2 and courtesy of Daniel P.B. Smith. https://en.wikipedia.org/wiki/File:TJ-2_memo_from_p2.jpg.

were also available in these formatters through markup commands. If one looks at research papers from the period, they are often formatted like this. The *nroff* program in Unix was a descendent of RUNOFF, starting from 1972.

After the invention of computer-controlled laser printing at Xerox PARC in the 1970s (Section 2.4.3) and other kinds of phototypesetting around the same time, formatting could be much more sophisticated. nroff for Unix evolved around 1976 into troff (for **t**ypesetter), which supported different fonts, proportional spacing, and other styling. Donald Knuth was unhappy with the quality of the resulting documents from troff, especially for equations, so he invented the TeX formatting language in 1978. In 1980, Brian Reid at Carnegie Mellon University created Scribe, which was significant in using semantic markup, like "heading" (instead of "bold"), and then defining how a "heading" looked in a separate place, which is how HTML + CSS works today. LaTeX was created in 1983 by Leslie Lamport (when he was working at SRI) to be an easier-to-use version of TeX and continues to be widely used today. All of these are actually fully functional programming languages, so users can write *macros* (a form of procedure or function) that define how the resulting document looks.

An intermediate design between formatting programs like troff and WYSIWYG programs like we have today was what were called *word processors* (also called *desktop publishing software*), which is defined generally by Wikipedia to be "a computer program or device that provides for input, editing, formatting and output of text," and thus is the same as the text editor–formatters described here, but the "word processor" term connotes systems that were *not* WYSIWYG but had a built-in editor with formatting commands, which were converted into (sometimes hidden) markup in the text. Probably the best-known stand-alone word processor machines were created by Wang Laboratories in the 1970s and early 1980s, and the best-known PC-based word processor programs include WordStar (see Figure 5.6(a)) and WordPerfect. These provided commands for formatting but typically only had character terminals so could *not* display the results on the screen. All of these word processing systems were eventually driven out of the market by Macintosh and Windows-based WYSIWYG editors.

As discussed above (Section 2.4.3.2), the researchers at Xerox PARC were inspired by the NLS system to make the first WYSIWYG text editor and formatter called Bravo (see Figure 2.11), around 1974. However, such interactive WYSIWYG editors did not reach the mainstream until the 1980s. Section 2.4.3.2 also relates the story of how Microsoft Word from 1983 came directly from Bravo. The Star (1981), Apple Lisa (1983), and Apple Macintosh (1984) all had multi-font WYSIWYG text editor and formatter programs included in their release, and they included support for laser and other printers that could print a higher-resolution version

of the pages as seen on the screens. High-end WYSIWYG editor–formatter programs were designed for printing professionals, who would write for magazines or books, and included programs like Interleaf (from 1981) and FrameMaker, which typically ran on personal workstations like the Sun (Section 2.2.2). These were sometimes called *document processors* to differentiate them from lower-level programs for consumers, like Microsoft Word for PCs and the Macintosh. Eventually, Word was augmented with sufficient features that it could handle whole books (like this one, which I am writing using Microsoft Word versions: Professional Plus 2013 and 2016 and then Word 365 Apps for Enterprise) and the workstation hardware disappeared, so there was a much smaller market for document processors.

8.6.2 Editing Systems for Code

Just as the first "users" of computers were programmers, the first use of text editors was for editing code, and this use continues today. Editors already mentioned above, such as vi and Emacs, saw their primary use as code editors, and many programmers continue to use plain text editors like these and Notepad for editing web page code (HTML and CSS). However, many specialized editors for code have been developed, typically called IDEs, as mentioned above in Section 8.4.2. Besides the usual text editing features, IDEs typically support automatic layout of the code by adding appropriate spacing and line breaks, moving the text cursor and selections by programming units, like going to the next statement, and "pretty-printing" the code by using colors and bold/italic to highlight comments, built-in keywords, strings, identifiers, and so on (see Figure 8.23). Generally, the formatting of code has been pretty simple (as shown in Figure 8.23), but some researchers have called for quite different formatting that emphasizes what programmers most need to know [Baecker and Marcus 1986], or for supporting graphics within the code [Ko and Myers 2006].

One unique IDE was the Smalltalk code "browser" from 1976 (note—not the same as a web browser), which had a hierarchical view to help programmers navigate to the desired method and see that method's code in the larger bottom pane (see Figure 8.34) [Tesler 1981]. Interestingly, this design, which focused on reusing the same window to show one method's code at a time, was the opposite of the design chosen for the Interlisp-D environment designed around the same time also at Xerox PARC, where each method was always shown in its own window, and the user needed to arrange and close all of these code windows manually [Teitelman 1979, Xerox 1983]. Whereas most modern IDEs have gone back to full-file editing rather than per-method approaches like these (usually using a tabbed-window style (Section 11.5), the Jasper [Coblenz et al. 2006] and Code Bubbles [Bragdon et al. 2010] research systems (and a Microsoft Visual Studio version they influenced)

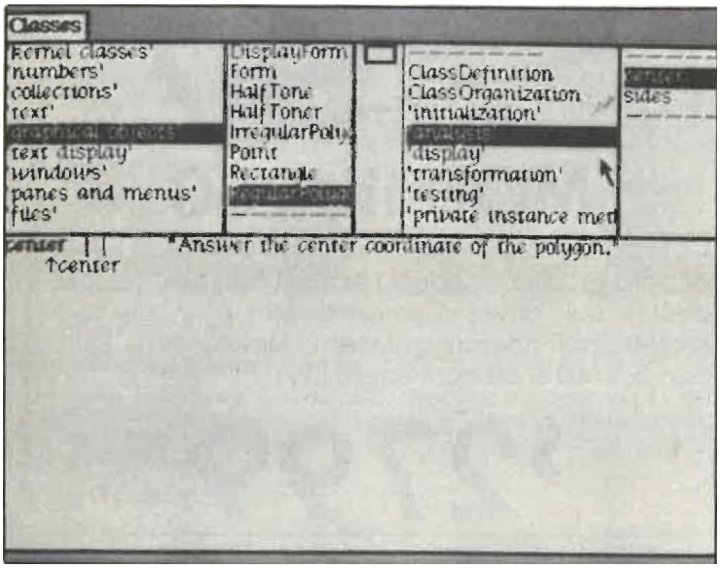


Figure 8.34 Smalltalk Browser for editing code—the top row shows the (from left to right) categories, classes, method categories, and methods, and the bottom pane shows the code for the selected method. Source: Photo 13c of Tesler [1981].

embraced having many windows each with a small amount of code, based on our research showing that programmers often need to view and edit multiple snippets located in various places as part of the *working set* needed to solve a problem [Coblenz et al. 2006].

Another difference between code and regular editing is that code has *structure*—which means that the "words" and punctuation (called the *tokens*) of code are limited and highly constrained and must be in a very specific order (called the *syntax* of the code). Some editors have taken advantage of this and help ensure that all typing and edits conform to the required structure. These *structured editors* (also called *syntax-directed editors*) do not support character-by-character text entry and editing but instead use menus, drag-and-drop, or single keystrokes that insert whole tokens or phrases (like typing "i" inserting "`if` *`condition`* `then` *`action`*"). One of the early famous structured editors was the Cornell Program Synthesizer from 1978 [Teitelbaum and Reps 1981]. We created related editors that tried to provide hybrid modes that supported some structure and some character-by-character entry and editing [Myers et al. 1988, Ko et al. 2005]. Today, some structured editors are still in use for environments for novice programmers so that they don't have to worry about syntax errors (e.g., Pausch et al. [1995] and Resnick et al. [2009]), but professionals still prefer the flexibility of regular character-level editing.

8.6.3 Text Editing Techniques

Similar to other interaction techniques like scrolling, text editing went through a phase of significant experimentation and diversity in the 1970s and then coalesced around the one interaction style used in the Apple Lisa and Macintosh, which continues to be the one consistent interaction style for desktops. However, the iPhone introduced a new interaction style for text editing on smartphones in 2007 that is significantly different from desktops and that has influenced all subsequent smartphone text interactions.

8.6.3.1 Moded versus Modeless Editing

The first decision for text editing is moded versus mode-free (or modeless) editing. I define a *mode* as meaning the same command does different things, depending on the mode. The classic example is that typing the "d" key might delete the selected item or enter a "d" character into the document, depending on whether the system is in command mode or insert mode. A classic story (see Section 23.8) about the problems of a moded system is to type "EDIT" at the top level of a command line interface, where this would enter the text editor, but if the user is already *in* command mode in the text editor, it would do **E**verything **D**elete, **I**nsert **T**, thereby deleting everything and inserting a "T" in its place, which was often a catastrophic failure since the early systems only had one-level undo, so it could only undo the insert of the "T" and not the delete!

The original NLS system was highly moded, as can be seen from the classic video—hitting the mouse button duplicates text, sets the cursor position, and various other things, depending on the mode (see Section 2.4.2). The Bravo text editor (Section 2.4.3.2) was highly influenced by NLS and was also moded—"r" put Bravo into replace mode, "d" into delete mode, "i" was for insert mode, and so on. The Esc key left insert mode to go back to command mode. As described by Larry Tesler (see Chapter 23), he saw people repeatedly making mode errors, which inspired his motto "DON'T MODE ME IN" (see Figure 23.3) and the development of the mode-free style of text editing we continue to use today. This same distinction applies to the competing editors vi/VIM, which is moded, versus Emacs, which is modeless.

8.6.3.2 Positioning the Text Cursor and Selecting Text on Desktops

Moving the *text cursor* and *text selection* are key interaction techniques for any text editor. As discussed in Section 4.2, the text cursor is where the next character will go, which may be different from the pointer cursor that follows the mouse or other pointing device (and touch devices like smartphones often do not have a pointer cursor). This section just discusses moving the text cursor.

Originally, on character terminals like shown in Figure 2.7, the text cursor was the same as the text selection and was *on* a character, which would be character to be deleted (see Figure 8.35(a)), but it was not clear where a *new* character would go—typically *after* the selected character. This same style was used in Bravo, where the selection was shown by underlining the selected character or characters. A click of the left mouse button selected a character, and the middle mouse button selected a word (see Figure 8.35(b) and (c)). Clicking in the left margin selected a line (left button) or paragraph (middle button). The selection was extended by clicking with the right mouse button [Myers 1990b].

Smalltalk introduced the kind of text cursor that is *between characters* since user studies showed that people were confused in previous systems about where new text would go. The Smalltalk cursor was shown as a vertical bar [Tesler 1981] or blinking caret *between* characters (see Figure 8.35(d)) so the location for next text was clear. The text cursor was placed by clicking the mouse button. The user selected text in Smalltalk by holding down the mouse button and dragging, also known as "press-and-drag." The selection was shown in reverse-video (light text on a black background). Another way to select text was to double-click in the same place to select words. Some systems support triple-click to select bigger units (sentences or lines in some systems, but triple-click selects paragraphs in Word and Google Doc). Four clicks might select even bigger units, or wrap around and just select a single word, as in Microsoft Word and Google Doc.

Smalltalk also introduced the interaction technique where any selected text is *deleted* when new text is typed, called *pending delete* (which could not be used in previous systems since at least one character was *always* selected) [Myers 1990b]. This style of text cursor positioning and text selection continues to be the standard interaction technique for all desktop systems.

Once the cursor is positioned, it can usually be moved, typically using the arrow keys on the keyboard. (As mentioned in Section 2.4.7, the original Macintosh did

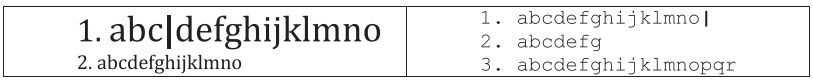
(a) (b) (c) (d)

Figure 8.35 (a) On character terminals, the text cursor is *on* a character (here "m"). (b) and (c) The Bravo text editor showed the selected text by underlining, here a capital "A" in (b) and the word "complete" in (c). (d) Smalltalk's text cursor was a caret shown *between* characters. Sources: (a) Screen capture. (b) and (c) Myers [1990b] around 1:02. (d) Myers [1990b] around 1:03.

not have arrow keys specifically to force people to use the mouse, including for moving the cursor position—see also Figure 2.16. The Xerox Star's keyboard also did not have arrow keys—see Figures 9.1 and 15.1.) Whereas left and right movements through text are straightforward, up and down are not when using variable width fonts. For example, in Figure 8.36(a), the cursor is after the "c" in line 1. When moving the text cursor down (e.g., with the down arrow key), all systems find the closest between character horizontal position to the current absolute position on the screen, and move there, which here would be between the "g" and "h" in line 2. Even with fixed-width fonts, when the cursor is at the end of the line, it is not clear where it should go when the next line is *shorter*, and then the line after that is *longer*. For example, in Figure 8.36(b), the cursor is after the "o" on line 1, and moving down will clearly need to put it at the end of line 2, but then where should it go for line 3? All current systems remember the original horizontal position and try to use that for vertical movement, so the cursor will move to after the "o" in line 3 as well. However, if the user had moved to the left when the cursor was on line 2 and then back to the right, this would override the remembered position, and the cursor would move directly vertically, to the "g" in line 3. Thus, there is an invisible mode.

Some systems have additional ways to move the cursor. For example, many systems will move the cursor by words by typing Ctrl-arrow (e.g., Ctrl-leftarrow). Sometimes, Ctrl-uparrow and -downarrow move by paragraphs. The Emacs editor has keyboard commands to move the cursor by many other units, such as forward by a sentence, or by a token or statement in a programming language. Ctrl-clicking on text that contains a link will often jump the text cursor to where that references. For example, in this book, all the cross references are hyperlinks to those locations since Ctrl-clicking in Microsoft Word will take me to what is referenced (and most of these links will be carried forward into the pdf version). Alt-leftarrow in Word moves back from a hyperlinked jump to the original position.

Once text is selected, there are ways to *extend* the selection. Systems since the Macintosh extend the selection when the user Shift-clicks with the mouse left button. Note that this typically extends from the *original position*, if the user Shift-clicks in different places, including switching sides of the original position (i.e.,



(a) (b)

Figure 8.36 Vertical movement of the cursor when (a) there are multiple fonts or (b) at the end of the line. Sources: Screen captures.

Shift-clicking below the original position and then Shift-clicking above the original position). Also, the selection extends in the same mode—if a word had been originally selected by double-clicking, then extending the selection will extend by full words instead of character-by-character. Similarly, if the selection mode had been lines, paragraphs, and so on, then that mode would be used for extending.

Since most people do not know about or just do not use the multi-click method to select words, Microsoft Word long ago introduced a "smart selection" feature that if the user drag-selects across a space, then the selection jumps to word select mode. I find this annoying since I often want to explicitly select part of one word and part of the subsequent word, so I turn this feature off (in the Advanced Options dialog box).

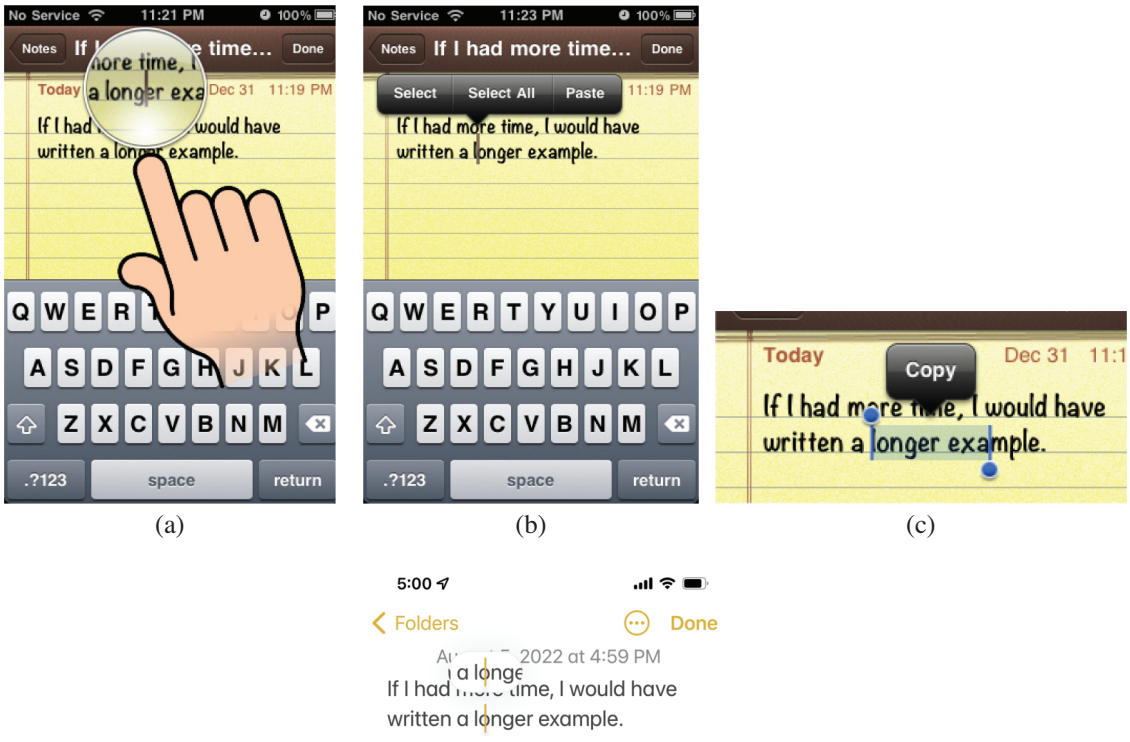
Although users obviously can select a number of adjacent characters in all text editors, most editors do not allow users to select multiple *independent* (non-adjacent) ranges of text. However, Microsoft Word does support this—select one range of text the usual way, and then use Ctrl-left mouse button drag to select an independent range of text. Operations like to change the color or delete the text will then operate on all the ranges, but other operations, like typing a new character (for *pending delete*), only affect one of the selections (currently, the last selection made, but it behaved differently in earlier versions), and all other selections are ignored and turned off.

Although it would seem like having the cursor between characters is clear, there are some complexities. For example, if the characters on each side of the cursor have *different styles*, like for "**blue**|*red*", what font should any new text have? Microsoft Word uses the style to the left (here **blue**), *except* when there was a character that was *deleted* at the cursor, in which case it uses the deleted character's style! There are also other complexities that are only revealed by experimentation.

8.6.3.3 Positioning the Text Cursor and Selecting Text on Mobile Devices

As discussed at length above (e.g., Chapter 3), early mobile devices were designed to work with a high-resolution stylus and thus generally mimicked the interaction techniques for mouse-based regular computers. For example, in Go Corp.'s "PenPoint" OS from 1991 (Section 3.5), a single tap on a character selected it, and a press-and-drag selected a range of text. However, PenPoint also supported drawing a circle around text to select more than one character, which is different from mouse-based techniques [Go Corporation 1992]. Microsoft PenWindows (Section 3.7) and TabletPC (Section 3.16) had the stylus emulate all of the mouse behaviors, so tapping placed the cursor between characters and drag-through was used for selection. The Palm (Section 3.13) worked the same way but did *not* support double-tapping to select words.

As with other interactions, the iPhone (Section 3.18) introduced innovations in how to select text, which was required by its use of low-precision finger pointing instead of a stylus. Putting a finger on the screen covers multiple characters that makes it hard for the user to see where the cursor will end up. Therefore, the iPhone provided a circular *magnifying glass* (sometimes called a *loupe* after a jeweler's magnifying glass) that appears *above* where the user's finger is, which shows a magnified copy of the text around the finger's press area, and the user can make fine movements to move the cursor while viewing the blown up area in the magnifying glass (see Figure 8.37(a)). Starting in iOS version 9 in 2015, the iPhone introduced "trackpad mode" where the user can press hard (called three-dimensional *(3D) press*) over the keyboard and the keyboard area changes into a trackpad, and



(a) (b) (c)

(d)

Figure 8.37 (a) Pressing on text on older versions of the iPhone displayed the text around the finger in a popup magnifying glass (also called a *loupe*). (b) Tapping twice in the same spot brings up the select buttons. (c) The handles for controlling the selection and the popup copy button. (d) iOS 15 in 2021 brought back the magnifying glass as a rounded rectangle. Source: Edited screen captures of iOS.

moving your finger around this area moves the text cursor correspondingly. Starting with iOS version 13 (2019), the magnifying glass feature (Figure 8.37(a)) was *removed*, and instead a long-press *in the text area* causes a cursor to follow the finger around, where the cursor is slightly above the finger press area (again so the user can see it since the finger is blocking part of the screen). Then, a version of the magnifying glass was brought back in iOS 15 in 2021 to show the cursor positioning, but now just in a rounded rectangle (Figure 8.37(d)). Meanwhile, as mentioned above (Section 8.3.6.2), iOS 13 also introduced Swype-like text entry when users move their finger around the keyboard, so the ability to use the keyboard area as a trackpad started to be available only when long-pressing on the space bar to start.

For selecting text, the iPhone had to use a different mechanism than desktops since press-and-drag moves the text cursor with the magnifying glass (see Figure 8.37(a)) and there is no Shift key to extend the selection. Therefore, the iPhone provides popup selection buttons (Figure 8.37(c)) that appear when the user taps in the same place twice (there is no time-out for this as for double-tap—the second tap can be any amount of time later). Another way to select text is to double-tap on a word, which causes it to be selected. The selection is shown with two handles at each end (Figure 8.37(c)) that the user can drag with their finger to adjust the size of the selection. As discussed in the next section (8.6.3.4), after selecting, editing operations appear as popup buttons (Figure 8.37(b) and (c)).

Subsequent mobile devices, such as Android, use similar interaction techniques as the iPhone for positioning the text cursor and selecting text, except without the magnifying glass.

8.6.3.4 Operations for Editing

Once text is selected, the user may want to perform operations on it. Although text editors have a lot of commands, the only operations are to *delete* the text, *change* the text into different text, or to *reformat* it (including its layout, font, color, etc.). As mentioned above in Section 8.6.3.1, a few systems support these operations in a *moded way*, so hitting a keyboard key like "d" by itself would cause deletion. However, most systems support modeless operations, so typing "d" enters a "d" character and deleting requires using the "Backspace" or "Delete" (to delete forward from the text cursor) keys on the keyboard or else using a Ctrl key like "Ctrl-D". On a mobile device, the virtual keyboard's backspace key (at the bottom right above the return key in Figure 8.37(a)) will delete the selected text.

Another feature for text editing is *drag-and-drop*—a way to move or copy text. Drag-and-drop can actually be supported in either moded or modeless systems—in the former the user hits a key to go into drag mode and another key to indicate that the cursor is at the desired destination. In "modeless" systems, the user presses

the pointer button down on text that has already been selected and drags while still holding the button down; the text will be deleted from the source location and will be moved (dropped) at the position where the button is released. If the keyboard's Ctrl key is held down before the pointer button is pressed, then the text is copied instead (the old text is not deleted and a new copy is placed at the drop point). Note that this is actually still moded, but the mode is indicated by holding down a physical button.

The cut, copy, and paste method for text editing was invented by Larry Tesler as a way to have a less moded alternative that would also work across different types of applications. The full story is described in Section 23.5, so here I just summarize the operation. On regular computers (and early mobile devices like the Palm), the user selects text and picks the "copy" command from the menu or accelerator key (usually Ctrl-C), which puts the text into the invisible clipboard buffer.[2] Similarly, the user can pick the "cut" command instead (with the accelerator Ctrl-X), which puts the selection into the clipboard and then deletes the original. The user can do arbitrary operations and then place the cursor in the target location, and pick "paste" (Ctrl-V) to cause the text to be put at that location. On the iPhone, cut, copy, and paste was first supported in version 3 in 2009 (see Figure 8.37(b) and (c)), but it remains awkward to use on phones and is not much used, especially compared to its ubiquity on desktop computers [Li and Myers 2021].

As a research project, my group investigated an interaction technique we called *wiggling*, which involves rapid back-and-forth movements of the mouse pointer (without pressing a mouse button) or up-and-down scrolling on a smartphone, to indicate that the text under the gesture should be selected, copied, and pasted into a note-taking application. An extra swipe in different directions at the end of the gesture can be used to indicate qualities about the text, like whether it is a positive or negative example. We found that wiggling helped participants in a user study accurately collect information with a 58% reduction in overhead while being 24% faster compared to a common baseline [Liu et al. 2022].

Another interesting aspect of text editing operations is *undo*, which is covered more fully in Chapter 15. The key issue for text editing is the *granularity* of the undo—how much typing each invocation of the undo command undoes. The easiest design is to just have undo remove each individual character or backspace typed, but this makes it very tedious to undo a lot of typing or deleting. Therefore, most modern editors undo a *block* of typing at a time, where the size of the block

2. On the original Lisa, the clipboard was actually visible—see the icon at the bottom middle of Figure 2.15(a). But Larry Tesler said that people rarely needed to look at it, so it was removed [Tesler 2011].

is context dependent—for example, separating by spaces, punctuation marks, the return key, switching from backspace to typing or vice versa, or moving the cursor. The Microsoft Word text editor has changed the granularity of undo across different versions to try to be more effective for users. Many operations are not undoable, such as saving the text to a file or moving the cursor, but some research systems have experimented with supporting cursor moves and scrolling in the undo stack [Myers 1998b]. A nice trick to know is that putting text into the clipboard is usually not put into the undo stack either, so you can undo a "cut" operation and the text will be undeleted—but the clipboard will still contain that text.

8.6.4 Multi-User Text Editing

When multiple users are editing the same document at the same time (as you can see today with systems like Google Docs), interaction techniques may need to work differently. This has been a long-studied area of what is now called *computer-supported cooperative work* (CSCW), or in general how technology can help multiple people work together. The original NLS system showed that this was possible (Section 2.4.2), but it was many years before examples were deployed. The simplest approach is that there is only one text cursor, and there might be a mechanism to control who gets to manipulate it. However, to support parallel work, often each user would have a separate cursor, and then some kind of control is necessary. An early approach was to lock out other users when one user was editing a section [Greif and Sarin 1987], but later systems provided more sophisticated mechanisms for managing multiple users' edits to the same region. A minimum requirement is to show the location of other users' text cursors and selection. The system may need intelligent ways to deal with overlapping edits to the same region of text, although even Google Docs today does not provide a robust interaction technique for avoiding or resolving conflicts but assumes users will communicate to avoid them (see also Section 15.10 about multi-user *undo* of text edits).

8.7 Text Entry Widgets

Text entry fields, also called a *text box*, *input box*, *text field*, *input field*, or *text entry box*, which allow users to enter characters, have been fundamental widgets since the first displays for computers appeared, as discussed in Section 2.5.1.1. They have continued to be important for GUIs and the web. The most basic type supports a single line of plain text (Figure 8.38(a)), with more elaborate ones supporting multi-line plain text (Figure 8.38(b)) or even formatted text.

Text entry widgets are supplied as part of user interface toolkits or libraries, and thus a programmer typically uses them without modification. Often, programmers

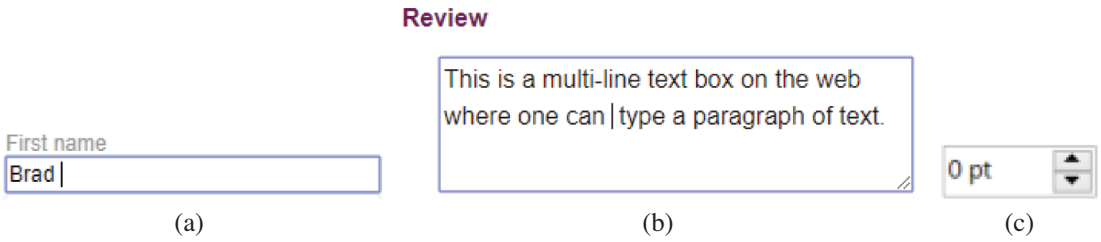


Figure 8.38 (a) Single line and (b) multi-line text widgets from web pages. (c) A spinner to the right of a text field for entering numbers. Source: Screen captures.

can configure whether there are restrictions on what the user can input—for example, only numbers, passwords (so the typed text is hidden by showing asterisks or dots instead), or text of a certain length. If only numbers can be entered, there may be increment and decrement arrows next to the text field (Figure 8.38(c)), which Wikipedia calls a *spinner* (which should not be confused with the iPhone *spin menu*; see Figure 7.7(b)).

If text is selected in the field, most fields today support *pending delete*, so that the text is deleted when new characters are typed. Often, a default or old value will be filled into the field before the user types anything (called auto-fill, Section 8.4.1), which may start off being selected. The user can typically move from field-to-field selecting which field has the *input focus* using the keyboard *Tab key* and backward to the previous field with Shift-Tab—the order is either set by the programmer or automatically from top-to-bottom, left-to-right (see also Section 9.10 about text fields in dialog boxes and Section 11.4 about the input focus).

The creator of a text field widget must also consider what editing options will be supported. Most fields support moving the text cursor and selection, usually with arrow keys and by clicking/tapping on the text, deletions using backspace and delete, and cut-copy-paste. Usually, the Enter (or Return) key causes a global operation, like submitting the form or dialog box, but sometimes it can be entered into the field (primarily for multi-line text widgets).

8.8 Typeahead

An important issue with any text entry method is *typeahead*—when the user can continue to type before the system is ready to process it, usually such that the characters the user is typing are not shown until later. As explained in Section 20.12.2, this is enabled at a low level by the event model queuing up the characters from the hardware or lower levels of the operating system and delivering them to the text widget or application when it is ready.

The advantage of typeahead is that the user can keep typing without worrying if the application is being slow, which might be especially helpful for very fast typists, or conversely for people who type looking at the keyboard so they might not even notice that the system isn't ready. The disadvantage is that users may accidentally type to the wrong place—for example, if something happens that was not expected.

A parameter when typeahead is supported is how much—is there a limit on the number of characters that can be queued? Another is whether *all* typing is queued—for example, are commands (like Ctrl-Z or Tab to move between text fields) included along with the regular keystrokes (usually most are). Operating systems often watch for some special keystrokes, like Ctrl-C in Unix or Ctrl-Alt-Delete in Windows, at a lower level of the operating system or hardware, so they can be used to abort an application that is hung and not responding.

8.9 Summary of Considerations for Designing New Text Entry Methods

As shown in Table 8.1, text entry speeds for mobile and wearable devices are still far from matching the speed of typing on a regular keyboard, and even that speed does not match the speed at which people normally talk. Therefore, there will continue to be a need for new and improved interaction techniques for text entry and editing. This section summarizes some considerations for designing new text entry and editing techniques, many of which were already covered in depth above.

- How much text will need to be entered? A few numbers versus a whole book.
- Text entry should be as fast as possible, preferably reaching typing speeds of about 40 wpm.
- Being reasonably accurate.
    - But considering how much accuracy is needed. For example, the difference between formal writing of a document versus text messages.
    - However, entering addresses and phone numbers may require high accuracy, even on a mobile device.
- Having a fast and accurate way to delete (repair) the entered text.
- Requiring little learning time for people new to the method but familiar with other methods (e.g., using a QWERTY letter layout for a new kind of keyboard).
- Providing accurate auto-fill, auto-correct, and autocompletion.

- Not causing fatigue, especially if people need to type a lot or for long periods. Avoid causing repetitive stress injuries and other harms (see also Section 17.3.1).
- Specialized requirements for the text—for example, is it code or mathematical formulas?
- Will typeahead be supported, and if so, for all or only some of the characters and typed commands?
- How does the user specify and change *where* the typing goes? The issues with changing which *windows* have the focus are covered in Section 11.4. Designers also need to consider how users specify the text cursor position and which text field has the focus.

8.10 Chapter Conclusions

Text entry and editing are probably the most studied interaction techniques, and there may be more guidelines for the design of forms and use of text entry fields than for any other specific part of the user interface. Still, it is the interaction technique with possibly the most innovation still happening, and the greatest need. As speech transcription continues to improve (discussed in Chapter 18), text entry using keyboards and other means may possibly decrease, but I don't believe it will ever disappear since people can handwrite and type to take notes when it would be inappropriate to talk, and speaking as a means of authoring takes away mental resources that are needed for planning the desired text.

Conventional wisdom when the Xerox Star was being designed 40 years ago was that executives would never type for themselves, since that is what secretaries were for, yet today, virtually everyone types text. It will be interesting to see whether that has changed 40 years from now.

CHAPTER 9

Forms, Property Sheets, and Dialog Boxes

9.1 Introduction

Full screen forms have existed since the advent of display screens, as described above in Section 2.5.1.2, and menus were used in many early systems (Section 7.4), but the idea of collecting together a diverse set of widgets into its own window seems to have originated with the Xerox Star from 1981 (Section 2.4.4), where they were called *property sheets*. The team at Apple that created the Lisa coined the term *dialog box* (sometimes spelled *dialogue box*) around 1983, which is the term that is still most used today. Other names that have been used include *option sheets* or just a *dialog*, as in an *alert dialog*.

9.2 Xerox Star Property Sheets

The property sheets on the Xerox Star were brought up by hitting the PROP'S key on the Star's special keyboard (see Figure 9.1) while an object was selected on the screen. This brought up the property sheet that was specialized for that object. One example was shown earlier in Figure 7.8. In addition to having widgets that looked different than today's, as shown and discussed above (e.g., Sections 5.6.4 and throughout 7.3), the Star supported a form of *tabbed dialog*, where a set of options would appear and disappear based on the selection in a higher-level widget. For instance, in Figure 7.8 (detailed in Figure 9.2), the selection of CHARACTER or PARAGRAPH in the top line controls which widgets are shown below that in the rest of the dialog. The Star designers motivated this design decision as "progressive disclosure" since they did not show all of the available properties at once [Johnson et al. 1989].

The property sheets in Star were generally *not* modal. This means that the window could be brought up and could stay visible while the user did other operations. For example, the user could select an object, use the PROP'S key to bring up its property sheet, change some values, and then use the "Apply" button at the top

Figure 9.1 Special keys on the Xerox Star keyboard, including the "PROP'S" key at the bottom right of the left group. See also Figure 15.1 for the right-side special keys. Source: Courtesy DigiBarn Computer Museum, http://www.digibarn.com/friends/curbow/star/keyboard/index.html.

of the window (Figure 9.2 shows the top of the property sheet of Figure 7.8). This would change the object but leave the window displayed so that it could operate on other objects. Alternatively, the user could click "Done" to apply the changes and dismiss the window (like "OK" in modern systems), "Cancel" to get rid of the window without changes (still called "Cancel"), "Defaults" to return the property sheet value back to their default values, and so on. The *menu* button at the far right of Figure 9.2 (now called the *hamburger icon*) brought up other commands as a popup menu (Sections 7.2 and 7.3.3). Other property sheets had "Start" and "Cancel" buttons, where "Start" corresponds to "OK". Another Star feature was a question mark graphic button in the title line that brought up help.

9.3 Apple's Dialog Boxes

Larry Tesler (Chapter 23) told me that the Apple Lisa team invented the idea of having dialog boxes appear for menu items that needed parameters—and using

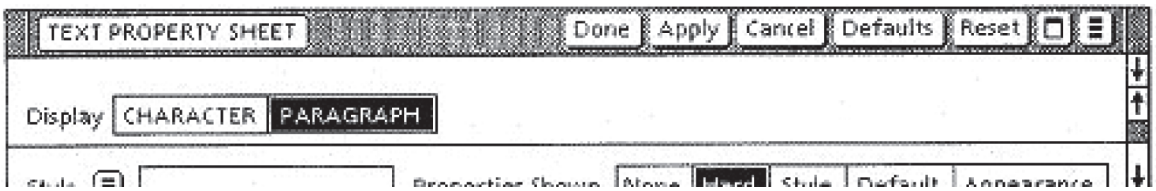
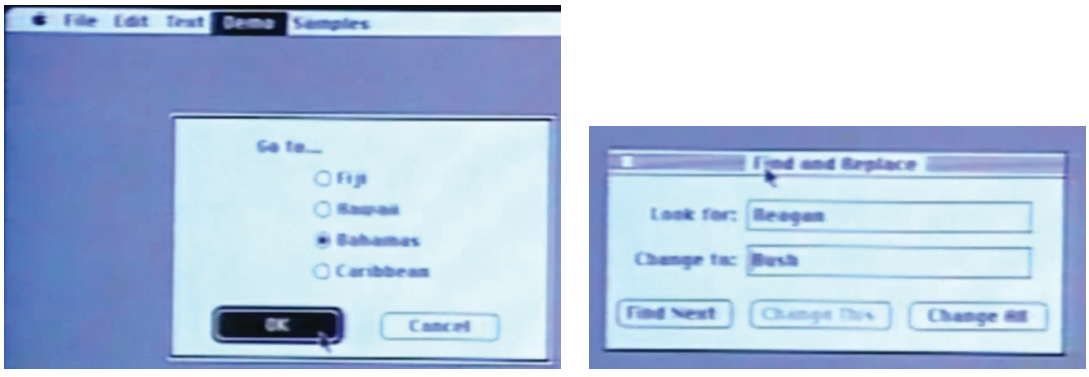


Figure 9.2 Detail of top of the property sheet shown in Figure 7.8. Source: Fig. 2 of Johnson et al. [1989]; see also http://www.digibarn.com/friends/curbow/star/retrospect/.

an ellipsis in the menu item name to signify this (see "Round Corners..." in Figure 5.11). That team also came up with the name *dialog box*. The Lisa was released in 1983 (Section 2.4.7), and the Macintosh (released in 1984) used the same design for these.

Apple's dialog boxes were either modal or not, where the modal ones originally could not be moved and would block all other actions until they were dismissed (modern modal dialog boxes on regular computers can usually be moved, and block all activity only for the one application, and don't prevent the user from switching applications). Most of the dialog boxes in the early Lisa and Macintosh systems were modal (Figure 9.3(a)). The key exception was the "Find" dialog (Figure 9.3(b)), which would find the next instance of a string in whatever window was in focus just before the Find window got the focus (see Section 11.2 about window focus). Modeless dialogs could be moved around using the title bar like any other window, but they usually could not be resized. The dialog of Figure 9.3(b) was closed using the window close box at the upper left.



(a) (b)

Figure 9.3 Dialog boxes on Apple Macintosh that are (a) modal and (b) not modal. Source: Myers [1990b] at 1:23:47 and 1:23:59.

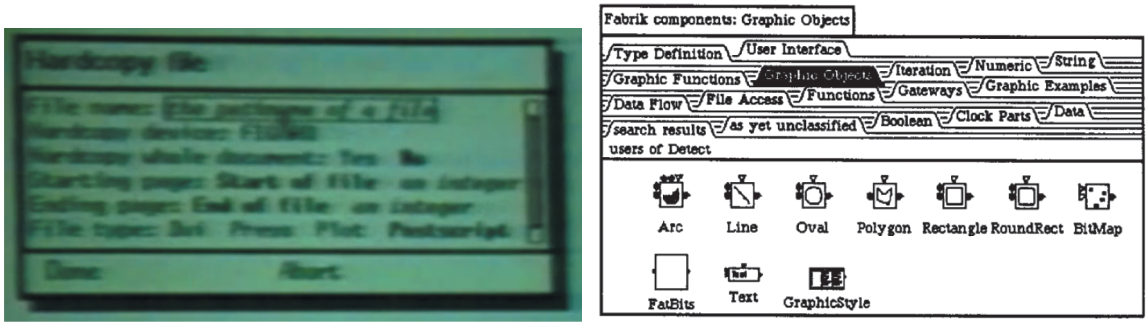
The Lisa and Macintosh also standardized on having the buttons that dismiss the dialog be at the bottom and usually were named "OK" and "Cancel". One of the buttons was the default (usually "OK" except for "dangerous" operations where "Cancel" was the default), which was shown with a darker border and would happen if the user hit the keyboard Enter key, as discussed in Section 7.3.8 and shown in Figures 7.12 and 9.3(a).

9.4 Tabbed Dialog Boxes

Originally, graphical user interfaces and applications for them had relatively few choices; they could all fit into the relatively simple dialog boxes in the previous section. However, as applications got more functions, the dialogs needed more content. One approach, used by the Symbolics Lisp machine in 1985 (see Section 2.2.2), was to just allow the dialogs to scroll (Figure 9.4(a)). However, this was not widely adopted.

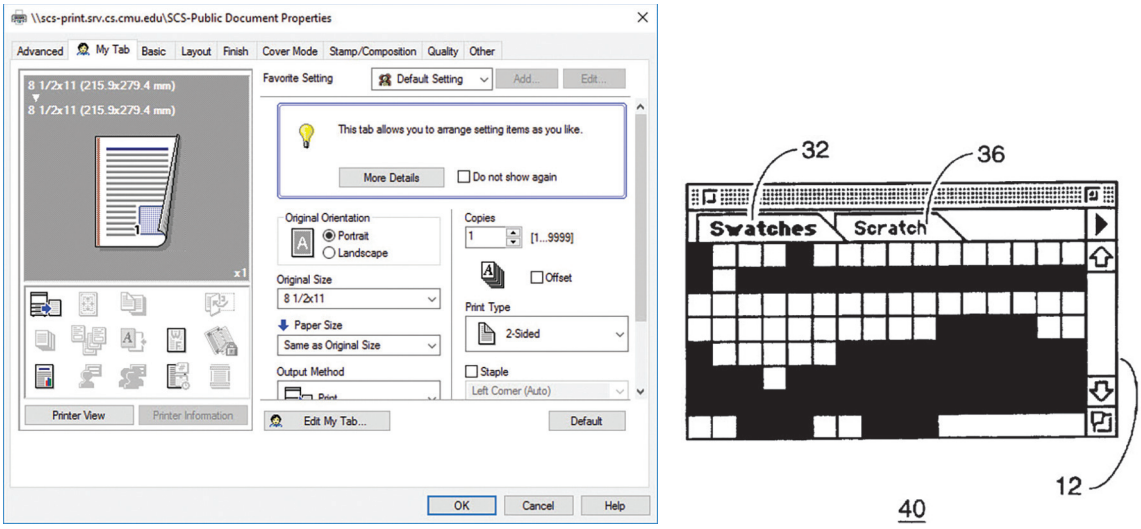
The idea of using notebook-like tabs when there are too many items to fit into one screen has been around for a long time. Section 11.5 below discusses using tabs for *windows*, whereas this section looks at tabs for dialog boxes. Although the Star had dialogs where different sets of widgets appeared based on the user's selection at the top as discussed in Section 9.2 (see also Figure 7.8, detailed in Figure 9.2), they did not look like tabs. The first use of notebook-like tabs in a dialog box that I am aware of is the Fabrik research system from 1988 [Ludolph et al. 1988], where the available elements were grouped by functions (see Figure 9.4(b)).

Microsoft Word version 6.0 in 1993 introduced tabs into the Microsoft Windows system for separating sets of properties in dialogs [Seymour 1994]; they



(a) (b)

Figure 9.4 (a) Symbolics dialog with scroll bar. (b) The Fabrik system from 1988 grouped elements into different tabs. Sources: (a) Myers [1990b] at 1:24:55. (b) Ludolph et al. [1988] Fig. 1a excerpt.



(a) (b)

Figure 9.5 (a) Modern Windows tabbed print dialog for Word 2016 in Windows 10. (b) Adobe's patent that describes draggable tabs. Sources: (a) Screen capture. (b) Johnston [1994] Fig. 7.

became a standard part of the Windows toolkit and are still widely used today (see Figure 9.5(a)). In 1994, Adobe submitted a patent [Johnston 1994] for multiple tabs for properties (see Figure 9.5(b)), which also supported dragging a tab out of the original dialog to be in its own separate window (much like tabs in modern browsers can be dragged out to be separate windows today).

9.5 Toolbars and Ribbons

As applications such as Microsoft Office began to include more and more features and commands in applications, it became slower to use the menus and dialog boxes to reach even the most popular commands and properties, so Microsoft introduced optional toolbars in Office for Windows 1.0 in 1989. A *toolbar* is a collection of buttons with small pictures on them (called *icons*) and sometimes also option menus and text fields, typically arranged at the top of an application window, below the menu bar if there is one (see Figure 9.6(a)). The intention is to provide a set of operations (*tools*) that the user can invoke quickly, typically with one click on the icon. The distinction between a *palette* (Section 7.2) and a toolbar is that palettes usually define persistent modes or options for the application (like what kind of object will be drawn), whereas toolbars have collections of commands as well as independent mode settings.

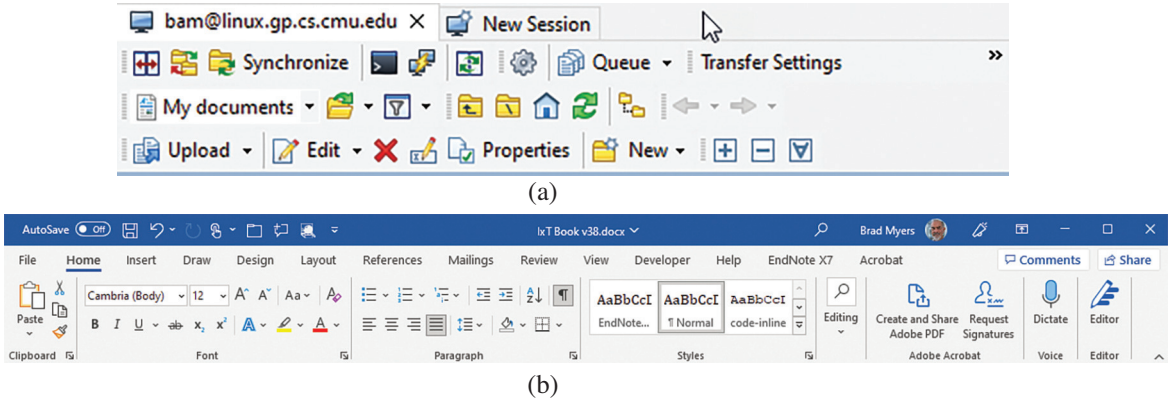


Figure 9.6 (a) Four rows of toolbars from Win SCP version 5.21.1. (b) The ribbon in Microsoft Word 365 version 2207 Build 16.0.15427.20182. Sources: Screen captures on August 8, 2022. Used with permission from Microsoft.

Since there were lots of commands in Microsoft Office and most other applications, there were *multiple* toolbars, and the user could configure which were shown and where, and often even which specific operations were in which toolbar. Unfortunately, this often resulted in there being many rows of toolbars, often with gaps since they were easy to move around by accident, and it was often tricky to get into the desired configuration. This resulted in much screen space being taken up by the toolbars (e.g., the four rows in Figure 9.6(a)). In Office 2007, Microsoft removed all of the toolbars and the main menu bar and replaced them with what it called the *ribbon*—a collection of tabs across the top of the window, where each tab opens a small space across the full width of the window containing commands and options using a variety of widgets (see Figure 9.6(b), where the "Home" tab is selected). Thus, the ribbon acts like a dialog box with a collection of related widgets. The ribbon originally did not support any customization but eventually allowed the user to remove and add some sections.

9.6 Typical Built-In Dialog Boxes

Most dialog boxes are designed for particular purposes by the application designer, but there have always been some dialogs supplied by the toolkits that applications could parameterize and use. For example, the original Lisa and Macintosh promoted the "Alert" dialog, for warnings or to confirm a dangerous operation by asking a yes–no (OK or Cancel) question [Myers 1990b, at 1:24:01], and the "File" dialog, which provided a standard way for applications to open and save files. Other commonly provided dialogs included ways for the user to select a color ("color-picker"), a font or the date and time, and to perform find-and-replace, print,

or print-setup. These are usually highly parameterized and may be provided in different forms even in the same toolkit.

9.7 Web Pages as Forms

Almost as soon as the Web was invented, Tim Berners-Lee and others recognized the need for pages to support data entry, so the first revision of HTML, called "HTML+" in 1993, provided "fill-out forms and input fields" [Berners-Lee 1993]. These supported a wide array of widget types including text and number entry, dates, URLs, checkboxes, radio buttons, integer ranges, "scribble" ("A field upon which you can write with a pen or mouse"), and buttons for "submit" and "reset" (see Section 10 of Berners-Lee [1993]).

A key innovation of the Web for forms is the way they provide automatic fill-in and autocompletion (see Sections 8.4.1 and 8.4.3) based on the browser's saved information and/or the server-side supplied values.

9.8 Dialog Boxes on Mobile Devices

Early phones with screens used a single column of fields for forms, such as changing settings or creating a new contact. Usually, the forms consisted of mostly text fields or "yes/no" toggles. PDA devices like the Palm Pilot (Section 3.13) had forms and dialog boxes designed to look like those on regular computers. Figure 9.7(a) shows a full-screen settings window with various widgets, and Figure 9.7(b) shows the popup dialog box. Other PDAs of that era were similar. The iPhone went back to a single column of widgets (Figures 7.9(d) and 9.7(c)), where some of the controls were arrows to take the user to another page of controls (e.g., "Language" in Figure 9.7(c)). The iPhone uses fewer text entry fields and more selections for settings, but it does have multiple columns for forms such as creating a new contact.

9.9 Research About Dialog Boxes

The early dialog boxes were designed and laid out using absolute coordinates. For example, the designer would pick the specific pixel location and size of each widget. There has been significant research, and then commercial tools, on allowing designers to do this through interactive tools (see Sections 2.5.2.2 and 20.6.3). Beyond simple layout, in 1992 my group investigated graphical tabs and graphical styles to help lay out the widgets consistently in an interface builder called Gilt [Hashimoto and Myers 1992]. Another feature of the same system was an easy way to declare dependencies among the widgets, such as when they were enabled and disabled or what the default values should be, which could depend on the values of other widgets [Myers 1991].

(a) (b) (c)

Figure 9.7 (a) Palm Pilot page of settings and (b) popup dialog box. (c) iPhone iOS v. 13.2 settings screen. Sources: Screen captures of Palm Pilot and iOS.

To avoid needing to do manual layout, some early UI toolkits provided the ability to declare that widgets should be in a particular position, such as a scrollbar that should stay attached to the right side of a window or the OK and Cancel buttons that should be centered at the bottom of a dialog [McCormack and Asente 1988]. The InterViews research system [Linton et al. 1989] adopted the *boxes and glue* metaphor from TeX to provide a different way to specify the layout of widgets and other UI elements (see Section 20.6.3.3). Meanwhile, Sketchpad (Section 2.4.1) had shown back in 1963 how general constraints could be used for placing objects, and this was investigated by later toolkits for widget layout (Section 20.6.2) [Szekely and Myers 1988, Myers et al. 1990, Hudson and Smith 1996]. However, these proved too expensive for the layout of widgets and unnecessarily flexible so that designers found them hard to use correctly [Myers et al. 2000a].

Meanwhile, other research systems investigated how to *automatically* select widgets and layout dialog boxes. For example, the Mickey system from 1989 took lightly commented Pascal (the programming language of the original Macintosh) variable definitions and automatically created properly formatted Macintosh menus and

dialog boxes [Olsen Jr. 1989]. For example, functions with parameters turned into menu items with "...", whereas Boolean variables turned into menu items that could be checked. The ITS system from IBM, also in 1989, provided a comprehensive declarative language in which all aspects of an interface could be specified [Wiecha et al. 1989]. My group's system (called Jade, from 1990) pioneered generating different looks from the same definition [Zanden and Myers 1990]. For example, the programmer could specify the choice of one-of-many or many-of-many, and the system would pick appropriate widgets (such as radio buttons or checkboxes, respectively) based on the target look and feel (see Figure 9.8(a) and (b)). Our group's personal universal controller (PUC) project from 2002 (see Figure 9.8(c)) showed that control panels for consumer electronics (like a copier or home entertainment system)

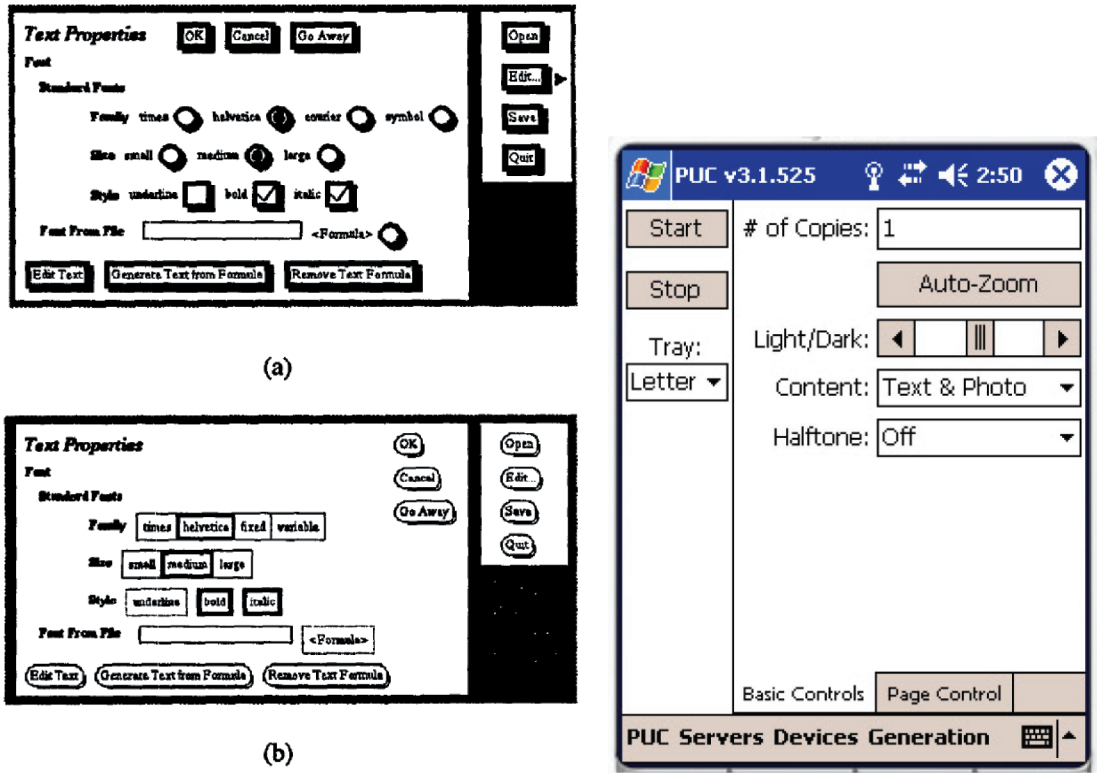


Figure 9.8 Jade automatically generated a dialog using the (a) Garnet style and the (b) OpenLook style for specifying the properties of a text object. (c) Control panel for a copier automatically created by PUC for a PDA. Sources: (a) and (b) Fig. 1 of Zanden and Myers [1990]. (c) Nichols et al. [2006a].

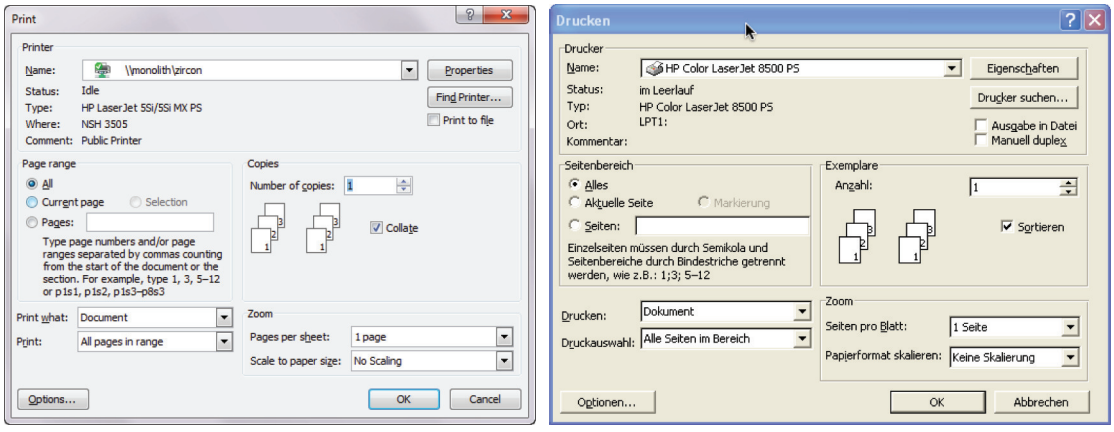
could be generated for a smartphone from a declarative definition and could be easier to use than the manufacturer's design [Nichols et al. 2002, 2006a]. A final example is the SUPPLE system from 2004, which used an optimization algorithm to determine the best layout for dialogs given constraints on the size and optional traces of typical user behaviors [Gajos and Weld 2004].

Other research relevant to dialogs has looked at more efficient ways for users to interact with the widgets in the dialogs. Making these more accessible has been a long-standing research theme, which is covered in Chapter 17. Other research has looked into allowing users to *gesture* or just to *cross* over the widgets rather than explicitly clicking on each one, which can be significantly faster especially when a user needs to change many values [Accot and Zhai 2002]. A project from my group, called Citrine, looked into more efficient copy-and-paste into forms, especially when *multiple* values need to be entered into multiple fields, such as for structured data like addresses [Stylos et al. 2004]. We have also looked into handling data transforms along the way—for example, if the source and destination fields do not use the same formats or groupings [Scaffidi et al. 2008].

9.10 Summary of Design Considerations for Dialog Boxes

In addition to the considerations for the widgets *in* the dialog boxes as discussed in previous chapters, there are a variety of additional considerations for the dialogs and forms, as discussed in this chapter. Here is a summary:

- Whether the dialog will be a *modal popup* or not. Modal dialogs freeze the underlying application until they are handled, whereas non-modal dialogs allow the user to interact with either the dialog or the underlying application.
    - For modal dialogs, the next decision is how much to freeze. In the original Macintosh (Figure 9.3(a)), everything was frozen, and the user had to dismiss the dialog before using this application or switching to another application. Today, however, applications cannot prevent the user from switching to a different application, so dialogs are only modal with respect to their own application. This can result in confusion—if the user can switch to a different application, and when switching back to the original application the modal dialog is sometimes no longer on top, the user tries to interact with the underlying application but every operation fails (typically, by just beeping).
    - For non-modal dialogs, the next decision is what the dialog is used with. For example, for Figure 9.3(b), *which window* is searched through?



(a) (b)

Figure 9.9 The print dialog for Windows in (a) English and (b) German. Note that the "pages" text is smaller and the buttons at the top right are wider. Sources: Screen captures. Used with permission by Microsoft.

  - Also, for non-modal dialogs, there may be subsequent interactions to cause the dialog window to stay on the screen rather than disappear after operating once. For example, the HyperCard menus could be "torn off" the menu bar and the OpenLook dialogs and menus could have a "pin" icon to cause them to stay on the screen (Figure 7.15).

- Whether the dialog is *movable*, *resizable*, and *scrollable*. Most modern systems for desktops allow their dialogs to be movable, whereas most for mobile devices do not. In contrast, most dialogs for desktops are *not* resizable or scrollable, whereas most web forms are resizable, and most web and mobile forms are scrollable.

  - If a form or dialog is resized, then how do the widgets in them react? Do they change size proportionally or reorganize? Web pages that reorganize based on the size are called *responsive*.

- Support for *internationalization*. Typically, the size of text varies in different languages, so a dialog would need to be re-laid out if it is translated into a different language (Figure 9.9).
- Whether to *gray out* or *remove invalid items* from the dialog. The same considerations apply as for menus items—see Section 7.3.4.
- How much is *canceled*? If the user hits "Cancel", this may require cascading the cancel up multiple levels. The classic example is if the user hits "quit",

but the file is not saved, an application might give the user a chance to save the file, but if the user cancels the file-save, this usually also cancels the quit as well.

- *Tab order*. The user should be able to hit the "Tab" keyboard key (or the "return" or equivalent key on mobile devices) to move to the "next" field, where the dialog designer should be able to define what "next" means. The input field that contains the text cursor is sometimes called the field with the *input focus* or the *focused field*.

9.11 Chapter Conclusions

Whether they are called forms, property sheets, or dialog boxes, these are still a key element of all systems since there almost always needs to be a way to enter, view, and edit a variety of information of different types. These are also beneficial in helping users understand specifically what information is needed in what order. However, a key aspect of conversational interfaces, where the user can use natural language, is that information can be provided in any order, at what would be different levels of dialog boxes. For example, it is easy to verbally say "enter the word 'apple' in the Heading2 style but using small caps aligned to the right margin," whereas this might involve at least three different dialog boxes and a dozen clicks if performed by direct manipulation. The future might involve various ways to combine input from various modalities.

CHAPTER 10

Creating, Selecting, and Manipulating Objects

10.1 Introduction

This chapter discusses interaction techniques to create, select, and manipulate objects, primarily graphical objects. It is a natural parallel to Chapter 8, which covers entering, selecting, and editing text. Although most people think of widgets when considering interaction techniques, for applications that contain more than forms, such as graphical editors, spreadsheets, visualizations, file browsers, and so on, the way that users interact with the main window *contents* is important too. Like with other IxTs, there are important considerations and design challenges for these interactions as well.

This chapter uses the word *object* quite broadly to include almost anything displayed to a user. This includes the graphical objects in a graphics editor like Adobe Illustrator or Microsoft PowerPoint, the text in a text editor or as a label among other widgets or graphics, the pixels in painting programs like Adobe Photoshop or Microsoft Paint, the files displayed using icons or using their names in a Microsoft Windows Explorer or Macintosh Finder window or iOS or Android main screen, and so on. *Graphical objects* will generally be the kinds of objects one would find in PowerPoint or a file browser (which does include text boxes), and I use this term when I want to differentiate from manipulating ranges of text. This chapter mostly focuses on two-dimensional (2D) objects—three dimensional (3D) is covered separately in Chapter 12. A discussion of the difference between *painting* and *drawing* programs is in Section 2.5.1.5.

10.2 Syntax of Operations

An important first consideration is the *syntax* of the interactions, or the order in which things must be done. In particular, is the operation specified first, compared to the parameters to that operation? There are basically three choices—named for where the operator is.

- *Prefix* means that the operator comes first. This is obviously the way that command line interfaces work (see Section 2.5.1.1), where the user types a command followed by the arguments, such as "`ren pic1.png ScrollBarPic.png`" to rename a file named "`pic1.png`" to the new name "`ScrollBarPic.png`" (see Figure 2.24). However, there are also some direct manipulation interfaces that use a prefix style, such as creating a new graphical object, where the user would typically select a rectangle icon from a palette in a drawing editor for "create new rectangle" (or in the "insert shape" menu in Microsoft Office), and then afterward would define the position and size of the new rectangle by dragging out a rectangle with the pointing device.
- *Postfix* means that the operator comes *last*. This is the most common order for direct manipulation interfaces, where the user selects an object (or a range of text), and then specifies an operation like "delete" or "change color". Postfix generally requires that there be a persistent way to show which object or objects are selected (as opposed to prefix, where there does not need to be a notion of a selection set—each operation can explicitly list its operands, like for command line interfaces). The advantage of postfix is that it is considered less moded since the user can select as many objects as desired and change which objects are selected—before specifying the operation. Since selecting the exact objects or text is generally more difficult to do correctly compared to selecting the operation, providing more flexibility for the parameter selections is preferable.
- *Infix* means that there are parameters on both sides of the operator. The classic infix operator is drag-and-drop, where the first parameter is the objects or text to start with, the operator is to drag them, and the final parameter is the target. Another example is the "format painter" command, which copies formatting—again, the user selects some source text or a graphical object whose formatting is to be copied and then hits the paint brush icon as the operation (shown at the left of the ribbon in Figure 9.6(b)). This puts the system into a mode where the formatting will be applied to the next location where the user clicks. Many of the operations on the Xerox Star were infix, which was part of the motivation for the copy-and-paste design of the Apple Macintosh—to avoid the inherent modes required for infix operators. Other infix operations include move and resize, which typically involve selecting the target object first (one parameter), then specifying whether to move or resize (the operation), and then specifying the new coordinates (another parameter).

Another interesting device where there was a big disagreement about the appropriate syntax for the user's operations was for calculators in the 1970s and 1980s (see Section 3.3). Texas Instruments made a line of calculators that used the conventional infix order for operators like + and *, and later models provided keys with parentheses for grouping. In contrast, Hewlett Packard calculators (see Figure 3.6) used *Reverse Polish Notation* (RPN) where all operations are postfix, which means that parentheses are never needed as long as operators always take the same number of parameters. For example, (3*(5-2)) has a different value (9) than ((3*5)-2), which is 13. In RPN, these would be 3,5,2,-* and 3,5,*,2,- since each operator uses the previous two elements and replaces them with the result, which can then be used by the next operator.[1] Interestingly, the PostScript language, created by Adobe to send documents from computers to printers [Adobe Systems Inc. 1985, Warnock 2018], which was also used to program window managers such as NeWS (Section 2.4.12), used a purely RPN syntax and was based on John Warnock's JAM graphics interpreter, which also was RPN. I used JAM to program my Incense system at Xerox PARC while John Warnock was still there [Myers 1980].

10.3 Graphical Object Hierarchies

Another important concept relevant to object creation and manipulation is what is called *2½ dimension* or *2½D* display of objects. This just means that some objects can be on top of other objects visually, so the objects or parts of objects that are behind are covered and hidden by the objects that are on top. In the memory of the computer, the objects are all in an order, from back to front, even if they are not currently overlapping. Some operations can bring objects to the front (in a graphics editor like PowerPoint, this is an explicit command "Bring to Front").

Many applications also have the idea of putting objects into a *group*. Again, in graphics editors there are explicit *Group* and *Ungroup* commands. In a file browser (like the Macintosh Finder or Windows File Explorer), a user can group objects by putting them into folders. Often, different kinds of objects can be put into the same group (e.g., rectangles and circles in PowerPoint and files and folders in the Finder).

Finally, objects typically have a *type* that cannot be changed. For example, in PowerPoint, one cannot change a rectangle to be a circle, and it doesn't make sense to change a file to instead be a folder.

1. In computer code, there are rules of *precedence* to carefully define the meaning if the parentheses are missing, so that, in all modern programming languages, 3*5-2 with no parentheses will be the same as ((3*5)-2) since multiplication (*) is given higher precedence than subtraction (-).

10.4 Creating New Objects and Painting

In graphical editors, including drawing and painting programs, there is usually a palette of the kinds of objects that can be created, which typically has been displayed on the left of the main graphics area. Usually, these include lines, various shapes, text, and freehand drawings (called *inking*). Examples of palettes include Xerox Draw (1975, Figure 2.12(b)), Apple LisaDraw (1983, Figure 5.11), MacDraw and MacPaint (1984, Figure 5.18), and all the way up to Illustrator today. Xerox Markup (1975, Figure 2.12(a)) used a popup menu with what could be painted, and PowerPoint now has the objects that can be drawn distributed throughout its ribbons. This is generally called the *tool metaphor* because the user first picks the appropriate tool (the draw-rectangle tool, draw-circle tool, etc.) and then uses that tool to create the object. This is *prefix* since the user selects the tool (the operator) and then uses that tool to specify the location. It might also be considered *infix*, if you consider selecting the color and other parameters part of the creating, when they can be picked before the tool selection.

Typically, for other kinds of applications, such as files in a file browser or data in a visualization, there will be application-specific creation techniques.

There are a variety of ways that the parameters for the new graphics are specified, such as where on the screen they will go and what color they are. Since these are similar or identical to the way the parameters are specified when modifying an existing object, they are all covered below in Sections 10.6 through 10.9.

10.5 Selection of Graphics

There are many considerations about how to *select* the objects on the screen and provide feedback to make clear to the user what is selected. In some applications, like a file browser, clicking on objects generally selects them. However, in graphical editors, the user typically has to go into *selection mode* (or leave drawing mode), often by clicking on the selection icon in the palette (e.g., the square in Xerox Draw (Figure 2.12(b)) or MacPaint (Figure 5.18) or the arrow in LisaDraw (Figure 5.11)).

10.5.1 Selecting One or Multiple Objects

Most graphical editors allow *multiple* objects to be selected through interaction techniques that *add*, *remove*, and *toggle* whether objects are in the *selected set*. For example, in PowerPoint and most other graphics editors, clicking on an object causes it to be the only object selected (deselecting everything else), whereas *Shift*-clicking on a different object that is not currently selected will cause it to *also* be selected. Actually, Shift-clicking is a *toggle*—Shift-clicking on selected objects causes them to no longer be selected.

10.5.2 Clicking on Objects

It may seem straightforward that objects become selected when one clicks "on" them, but for many objects this is not well defined. The simplest way to implement this is to count clicking anywhere in the object's *bounding box* (the smallest rectangle that fits around the object), but this is a bad idea for many objects, the worst being a diagonal line, which fills very little of its bounding box (see Figure 10.1(a)). Another issue is objects with holes or see-through parts, like an ellipse with an outline that is *not* filled. For example, clicking where the arrow points in Figure 10.1(a) should probably select the gray square and not the red ellipse or the green line. Similarly, objects that are behind and covered (in 2½D) by another object at the click point should not be selected either, like the blue triangle in Figure 10.1(a). However, for very thin lines, it can be difficult for users to click directly on them, so many systems allow clicking in a region around the outline of the line.

To remove the selection (so nothing is selected), the user can usually click in the background where there are no objects, or Shift-click on the only selected object.

When an object is part of a group, clicking on the object usually selects the group instead, but some systems allow the individual object *and* its group to be selected, with different operations affecting one or both. For example, PowerPoint allows the text of a label object to be edited even while that text is part of a group, by clicking on it twice (the first click selects the group and the second click selects

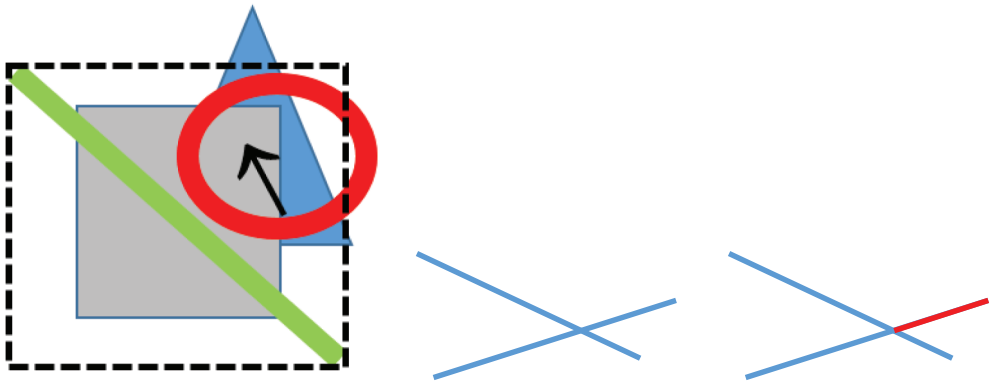


Figure 10.1 (a) The bounding box of the green line is the whole dotted rectangle. Clicking at the arrow should cause the gray rectangle to be selected, not the line or the red ellipse, even though both are on top of the square, and also not the blue triangle, which is behind the gray square. (b) The intersection of two lines can define four segments, (c) so each could be individually edited. Sources: Original by Author.

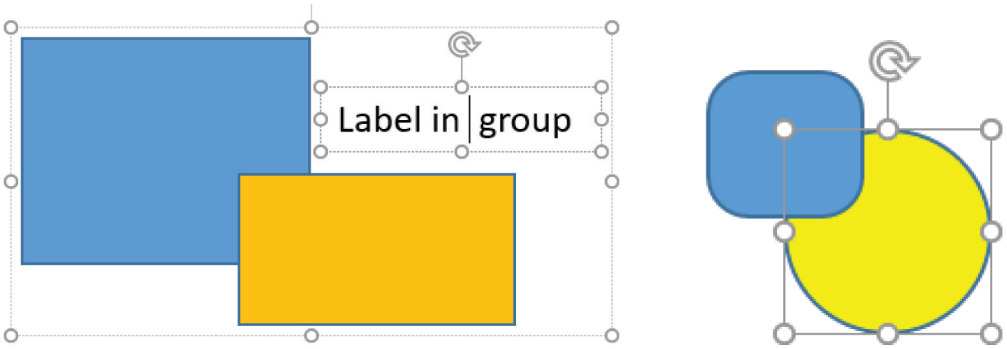


Figure 10.2 (a) These objects are in a group, which is shown by the dotted outline and handles on the outline. The text label is also selected and is editable while still in the group. (b) Selection handles show through objects on top. Sources: Screen captures of Microsoft PowerPoint 2006. Used with permission from Microsoft.

the label in the group (see Figure 10.2(a))). Other graphical editors provide various ways to *drill down* into the elements that are part of the group. Related to grouping is the concept of *layers*, which are used heavily in Adobe Photoshop and other editors—a *layer* is a collection of objects and/or paint that can be separately turned on and off (along with other operations).

Another variation on selecting objects is the ability to decompose objects into sub-pieces. For example, Adobe Illustrator allows a rectangle to be decomposed into its four lines, and then these can be separately edited. Even harder are segments defined by the intersection of independent objects, such as for two lines that cross, allowing the user to select and edit any of the four emergent segments (Figure 10.1(b) and (c)).

10.5.3 Selecting in a Region

To select a collection of objects, most systems allow a user to drag through multiple objects to add them all to the selection set, or remove them if they were already selected. This usually begins by pressing down the mouse button in a blank area and then dragging out an area. Assuming nothing was selected to begin with, then all the objects that are fully inside the area will become selected. Since this operation is usually a toggle, if anything inside the area had previously been selected, it will become unselected. This operation usually ignores the 2½D layout, so even fully covered objects that are inside the area will be affected. It will even select invisible objects that might be otherwise difficult to click on.

Some systems can select or deselect any objects that *intersect* the area dragged out by the pointer, instead of requiring those objects to be fully *inside* the area. This may be the default operation, or else the user might be given the choice of how bounding-box selection should work.

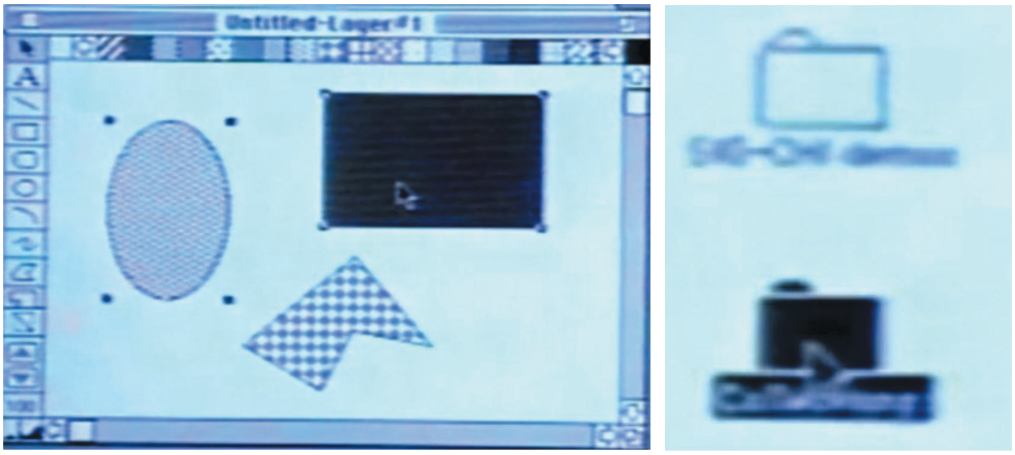
10.5.4 Feedback for Selecting Objects

The Sketchpad system from 1963 (Section 2.4.1) demonstrated many of the features of drawing and other direct manipulation programs, but it did not have the concept of a selected set of objects. Instead, the operations just affected the single object at the current pointer location. The Draw program from Xerox PARC from 1975 (Sections 2.4.3.3 and 2.5.1.5) displayed small squares on the selected objects (see Figure 10.3(a)), but these were not "handles" in that the user could not click on the squares. Similarly, on the Xerox Star (1981) multiple squares were used on the pieces of segments to show the selected objects (see Figure 10.3(b)).

The next key change in the way that the selected feedback was shown was in the Apple Lisa and Macintosh, which pretty much defined the method we still use—selection of graphical objects is shown by sprouting square *selection handles* and non-graphical objects by reverse video (using white on a black background instead of the reverse), as shown in Figure 10.4. A key difference on the Apple systems from the prior ones just mentioned is that the squares *are* handles, since pressing and dragging on a handle changes the object's size whereas pressing on the object but *not* on a handle moves the object. As file system objects cannot be resized, no handles are required, and reverse video is sufficient (Figure 10.4(b)).

(a) (b)

Figure 10.3 (a) Close up of small squares used to show the selection in Xerox Draw from 1975. (b) Selection feedback on the Star from 1982. Sources: Myers [1990b] at (a) 1:30:56 and (b) 1:31:27.



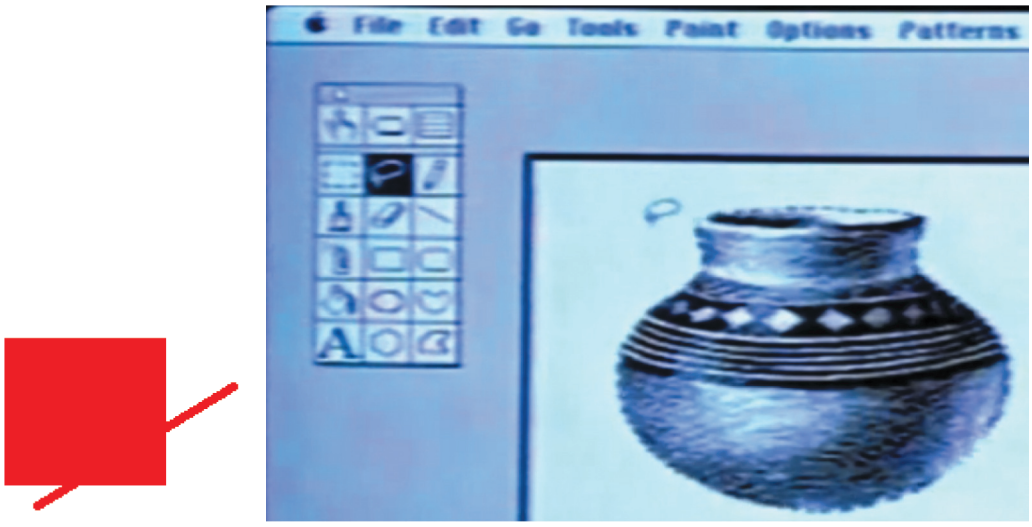
(a) (b)

Figure 10.4 Selection in the original Lisa and Macintosh used (a) handles and (b) reverse video. Sources: Myers [1990c] at (a) 1:32:55 and (b) 1:31:58.

Some systems have augmented the feedback of what is selected. For example, PowerPoint now uses circles instead of squares as the handles (Figure 10.2(a)) and also sprouts a *rotation handle* on the top of the selected object(s). If groups or non-rectangular objects are selected, the feedback includes a dotted rectangle around the bounding box of the selected objects, and handles on the outline (Figure 10.2(a) and (b)), as well as any selected objects in the group. Another minor variation is that the selection handles show through objects that might be covering the selected object, so the user can always see where the selection is even if the selected object is partially or fully covered (Figure 10.2(b)).

10.5.5 Selecting Pixels for Painting Programs

As discussed in Sections 2.4.3.3 and 2.5.1.5, drawing programs have individual graphical objects whereas *painting* systems have only dots of color (pixels) on the screen. Typically, the system will provide various kinds of "paint brushes" with which the user can change the color of pixels in the image. Once the paint is applied, however, the integrity of objects is lost. That is, if the user paints a line after the pixels of the line are painted onto the image, the fact that they came from a line brush is lost. Therefore, users would not be able to select that line anymore. This is especially apparent if a colored line is painted over a background with the same color (see Figure 10.5(a)). Note that all modern systems and many early systems were actually hybrids—for example, allowing textual objects (that retained their integrity after painting) along with the paintings.



(a) (b)

Figure 10.5 (a) When a red line is drawn over a red square in a painting program, the integrity of the line is lost. (b) The lasso tool and "marching ants" selection feedback in the original Macintosh MacPaint. Sources: (a) Original by Author. (b) Myers [1990b] at 1:33:35.

Early systems like Draw and SuperPaint from around 1975 (Section 2.5.1.5) provided selecting rectangles of pixels that could be copied, cleared, filled with a fixed color, and so on. Typically, the feedback of what was selected was just shown by inverting the pixels (changing the color of the pixels). Apple introduced a novel way to show the selected pixels in MacPaint (1984) through the *marching ants effect*, where a dotted line is shown around the selection and the dots animate ("march") along the line (see Figure 10.5(b)—but it is hard to capture this in a still picture). This makes the selection visible independent of the background. MacPaint provided a "lasso" tool (selected in the palette in Figure 10.5(b)) with which the user could draw a freehand shape, and the tool would shrink the selection to the non-white pixels.

10.6 Specifying Position, Size, and Shape

In some applications, objects are created in a specific place and size, but in many other applications, the user must specify the position, size, and shape of objects. After they are created, objects usually can be selected and then moved, and their size changed by the user. For some objects, such as polygons and curves, the shape can also be specified and modified. This section focuses on the properties of objects that are defined by coordinates, and the next section discusses the other properties like color, line thickness, and so on. Also, this section just addresses 2D

objects (actually 2½D since they are on top of each other)—3D objects are covered in Chapter 12.

10.6.1 Coordinate Systems

When specifying the position and size of objects using interactive interaction techniques, the user can just point to the desired location. However, behind the scenes, the implementation must keep track of the coordinates of that location. Often, these coordinates are visible to the user, either due to gridding (Section 10.6.5) or just so the user can more precisely know or adjust the location.

The most obvious way to define the coordinates is in pixels—the number of pixels across or down the screen. This is clear and precise but does not necessarily help with what the user sees since screens with the same number of pixels can be vastly different sizes. For example, an iPhone 11 Pro from 2019 has a screen resolution of 1125 × 2436 pixels for 2.81 × 5.67 inches, or about 400 pixels per inch, compared to a Microsoft Surface Hub 2S that has 3840 × 2560 pixels for 43.2 × 29.2 inches, or about 80 pixels per inch. So, a circle that is 100 pixels in diameter will be ¼ of an inch high on a phone and 1¼ inches high on the Surface Hub. Nevertheless, images will usually be specified by their width and height in pixels.

Another popular size is *points*, as in the point size of fonts. A "point" is 1/72 of an inch, so 12 points is about 1/6 of an inch. Although defined in *inches*, this does not necessarily define how large the characters will be displayed on the screen since the user can control the scaling factor and screens are all different sizes. However, programs will generally display items at a normal zoom factor, and this is the size at which it will usually be printed. Finally, some systems work in inches or centimeters, but again these are only approximate with respect to the size at which the objects will actually be displayed.

Another issue with coordinate systems for locations is—where is the origin? When a programmer draws a graphical object, the origin is usually at the *top-left* of the window, with larger numbers going down and to the right. However, many drawing programs will provide the user with coordinates starting at 0,0 at the *bottom*-left, with larger numbers going *up* and to the right. Negative locations are usually supported for objects that start outside the boundaries of the displayed area.

10.6.2 Global Parameters—Feedback and Aborting

Across all kinds of operations on graphical objects, there are global characteristics about how the operation works. This section covers the issues of feedback and aborting.

Feedback for the operations shows the user what is happening. For example, when creating a new object, or moving or resizing it, the user needs to know where it will go. I call the feedback one sees while the operation is in progress the *interim feedback* (related to *interim selected* discussed in Section 7.3.11.1) and the feedback to show the final result of the operation the *final feedback*. The final feedback for all operations is straightforward—showing the object in its new place with its new properties. The Sketchpad system from 1963 (Section 2.4.1) was arguably the first system to provide graphical interim feedback with what we now call *rubber band lines*. However, the Xerox Draw editor from 1975 did not provide any interim feedback [Myers 1990b at 1:35:13], presumably because the processor was not fast enough. The graphical editor on the Xerox Star provided no interim feedback for moves but showed the actual object itself changing for resizing [Myers 1990b at 1:36:10]. The Apple Lisa and Macintosh provided dotted lines as interim feedback and left the original object in its original place (for moving and growing). This style of interim feedback was widely used until processors became fast enough to always change the original object itself in real time, which is today's standard form of interim feedback.

A related characteristic of operations is whether they can be aborted or undone. As discussed in Section 1.8.3 and Chapter 15, the first question is whether the user can *abort* (or *cancel*) the operation *while it is in progress*. For example, on Microsoft Windows and in Microsoft Office products, virtually every operation can be aborted by hitting the "Esc" key on the keyboard, while the mouse button is still being held down, to make the operation not happen (and then the mouse button release is ignored). After an operation is finished, virtually all systems support *undo*, which causes the operation to be undone (see also Chapter 15).

10.6.3 Size and Position for Bounding Boxes

For objects that are created using a bounding box, which includes rectangles, rounded rectangles, ellipses, and various other shapes, the user typically can press down with the left mouse button to define one corner of the bounding box, then drag with the button down, and release for the other corner. Just clicking (press and release at the same point) sometimes creates a zero-size object (which is pretty useless) or aborts (and doesn't create anything) or creates an object with a default (non-zero) size. A key parameter to the creation operation is a minimum size, and dragging out a bounding box smaller than this size will either abort or use the minimum size.

Another parameter controls what happens if the user drags the second corner to the opposite side. For example, the user might press down at the upper left corner and drag down and to the right, or else the user might drag *up and left* from the

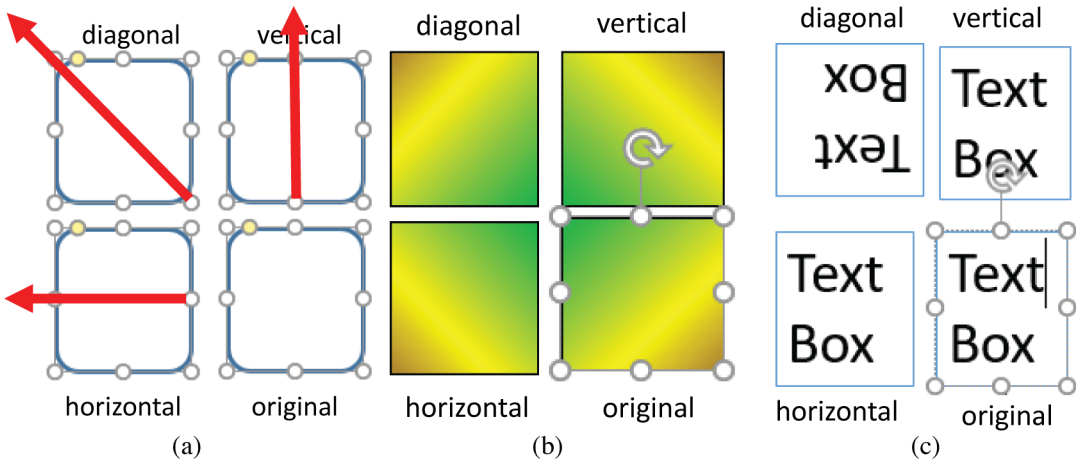


Figure 10.6 In PowerPoint 2016 (v. 16.0), pressing on the handle shown at the base of an arrow in (a) and dragging in the arrow's direction across the opposite end results in different flipping for (b) graphics versus (c) a text box. Sources: Screen captures, used with permission from Microsoft.

original point. Usually, this flips the object to be on the other side of the original point, but it may also flip the object upside down and/or backward—for example, if the object is filled with a pattern or color gradient or contains text (see Figure 10.6).

In modern systems, objects are usually resized using handles that are placed around the selected object—dragging on a handle changes the size from that corner or side. Again, there are issues about whether the user can make the object be zero size or smaller than some minimum while resizing. In some applications, making an object be zero size is allowed, and then the object usually cannot be seen and can only be selected by drawing a bounding box around where it is. Other times, making an object be zero size results in it being deleted or else causes the object to peg at a minimum size.

There is also the issue of whether the user can cross over and switch sides (e.g., if resizing from the right edge, can the user drag the edge to the *left* of the left edge to flip the object horizontally), and if allowed, does this flip the contents. I call this *flip-if-change-sides*. For example (see Figure 10.6), PowerPoint 2016 (v. 16.0) flips the fill direction for graphics if one drags past the original point in either direction or diagonal, and will flip the text vertically if one drags above the original point on the diagonal (so the text is upside down), but it does *not* flip the text horizontally if one drags past the left (so the text is *not* mirror imaged). Even more confusingly, the outcome depends on the order of the edits. This behavior has been different in different versions of PowerPoint.

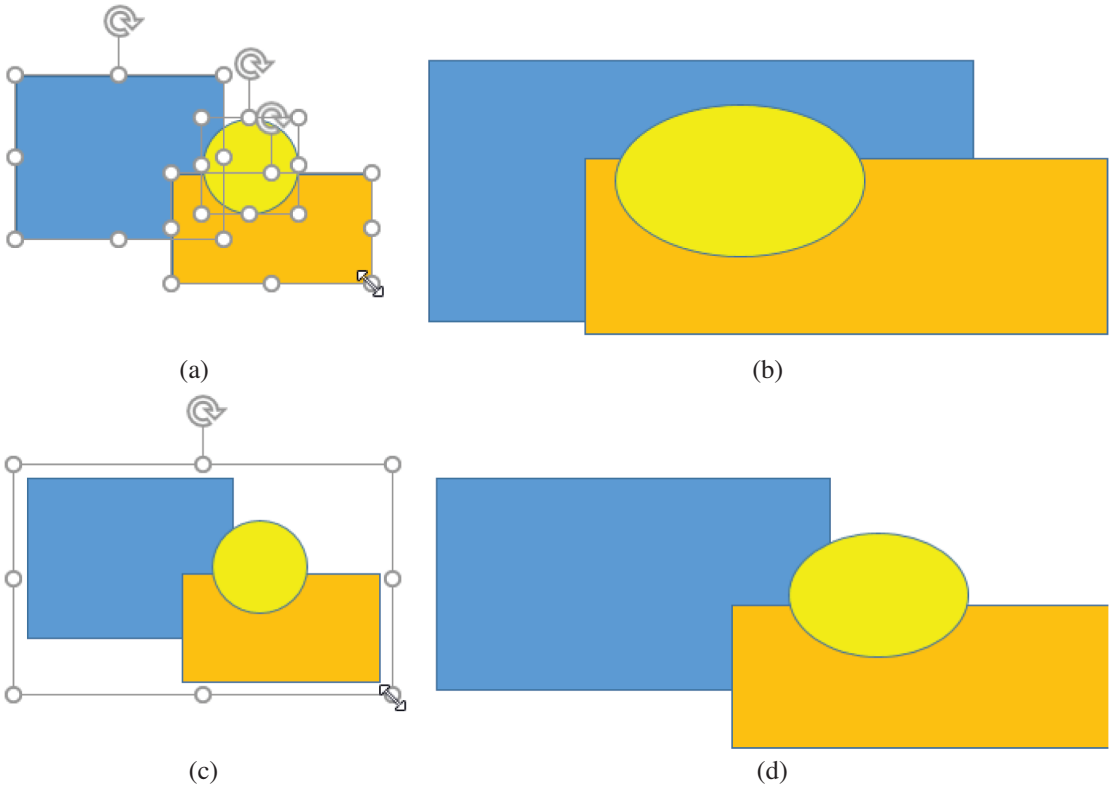


Figure 10.7 In PowerPoint 2016 (v. 16.0), selecting multiple objects separately (a) and resizing them all using the handle on the bottom right one, (b) resizes each from that object's upper left corner (so they all stay at their original places). However, grouping the objects (c) and resizing the group (d) affects each object's size *and position* proportionally. Sources: Screen captures, used with permission from Microsoft.

Often, holding down a modifier key, such as the Shift key on the keyboard, will affect how the sizing works. For example, when newly creating an object, holding down the Shift key will often constrain the size to be square, so a perfect square or circle can be drawn (rather than a rectangle or ellipse). When resizing an existing object, typically this modifier will keep the original aspect ratio, so the shape will not change while resizing.

There are various options about how *multiple* objects are resized. For example (see Figure 10.7), in PowerPoint 2016 (v. 16.0), if multiple objects are selected and the handle on one of them is pressed and dragged, then each of the objects resizes proportionally from that object's own origin—so they don't move. However, if the user *groups* the object first and resizes the group, then the behavior

is quite different—they all resize within the group so the objects both move and resize.

For *moving* the selected object or objects without resizing, modern systems since the Lisa and Macintosh have all allowed the user to press anywhere inside the object that is *not* on a handle to move objects. Often, for tiny objects it can be tricky to find a spot that isn't covered by a handle. Another issue is open objects, like a circle that is not filled, where the user must press on the outline in order to move the object (and not in the open inside of the object). In PowerPoint 2016 (v. 16.0), this also applies to selected groups—the user must still press on a graphical object in the group (and *not* just in the bounding box of the group that is outside the objects yet inside the group's outline) in order to select or move the group.

Most graphical editors allow objects to be *nudged* (moved by a small amount, such as by one pixel) using the keyboard arrow keys. This can be very useful to get objects into the exact desired place. Usually, the nudging takes into account any defined grid (Section 10.6.5) so that when a grid is defined each press of the arrow key will move by one grid unit. Note that when the selected object is text, there is a conflict about whether the arrow key should move the object or move the text cursor inside the object (Section 8.6.3.2). Typically in graphic editors, the user must click multiple times on a textual object to cause the text cursor to appear and thereby change modes from graphical editing to text editing.

An independent consideration is that objects with thick edges expand on *both sides* of the origin and destination point. That means that if the outline of a rectangle is 5 thick (in any unit), and the position is at 10, then the edge will actually extend from 8 through 12—going on *both sides* of the origin point (see Figure 10.8). The same is true for drawing thick lines, curves, or scribbles. The Adobe Illustrator program provides a mode that makes thick lines grow only to the inside rather than on both sides.

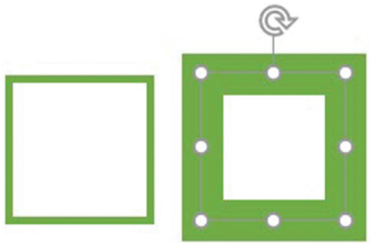


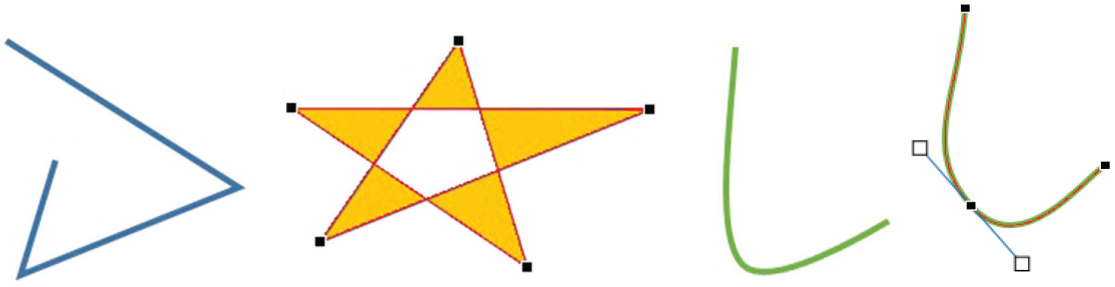
Figure 10.8 Thick lines go on both sides of the origin point. These two squares have the same "Y" coordinate, but the selected one on the right has a thickness that is 5 times the one on the left, so it goes 2 in each direction. Sources: Screen capture from PowerPoint 2016 (v. 16.0), used with permission from Microsoft.

10.6.4 Size and Position for Lines, Curves, Polylines, and Freehand Scribbles

Defining the position for line-like objects (including lines, arrows, etc.) is simpler in that the user is always allowed to start and end the drawing in any direction, and editing an endpoint is usually supported by clicking on either end. Most systems also allow lines to be moved without changing their orientation and length and sometimes also support resizing without changing the orientation. As with bounding-box editing, usually holding down a modifier key, such as the Shift key on the keyboard, causes the line to only be exactly horizontal, vertical, or diagonal.

For objects created by a sequence of lines, such as polylines (multi-line open shape, see Figure 10.9(a)) or polygons (multi-line closed shape, Figure 10.9(b)), the key interaction technique question is how to stop. For example, one can click and then click again to make the first line, with each subsequent click creating a new line segment. If the user clicks close enough to the first point, then this will usually stop the creation and define a closed polygon. To create an open polyline, the user might double-click to stop or hit some other key. To edit an existing polygon or polyline, there is usually a command that puts handles at the line intersections (called *Edit Points* in PowerPoint) that can then be adjusted. There may be an explicit command (or double-clicking) to delete that point or to add a new point along a line.

A curve in computer graphics, sometimes called a *spline* or *Bézier curve*, is mathematical way to draw a curve that is defined by specific points, sometimes called *knots*. A more complete description can be found in any graphics textbook (e.g., Newman and Sproull [1973] and Foley et al. [1990]), but here we focus on how users can create and control a curve, which is often defined by three or more points. These might be entered and edited as with a polyline (see



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

Figure 10.9 (a) Polyline defined by 4 points. (b) Self-intersecting polygon defined by 5 points. (c) Curved spline from 3 points. (d) Spline control handles. Sources: Original by Author.

Figure 10.9(c)). An additional control for curves is a tangent line with its own handles (see Figure 10.9(d)) that can be manipulated to change the curvature around the attached point.

In a painting program, a *freehand curve* or *scribble* (sometimes called *inking*) is left as the pixels that are drawn, but in a graphical drawing program, freehand curves are generally turned into a polyline or sequence of curves so that they can be scaled and edited, and printed at a higher resolution than drawn.

10.6.5 Grids

Graphics applications provide a variety of interaction techniques to help users place and size objects more precisely. These generally can be described as *gridding* or *snapping*. A *grid* is a regular pattern of locations on the screen that points of an operation are constrained to use. For example, a grid of 10 pixels would mean that the location would have to be at a multiple of 10. (Note that the grid might be expressed in pixels, points, inches, or centimeters, depending on the coordinate system, as discussed in Section 10.6.1.)

Designing a gridding system has many complexities. The first is the *origin* of the grid—where is 0,0? Typically, this would be at the origin of the coordinate system, typically at the bottom or top and left of the window. However, it is also possible for a grid to depend on a reference line or axes, so that objects stay aligned with each other rather than with the border of the window. When *moving* an object, usually the user can click anywhere inside the object to move it, but it still should be the origin corner that snaps to the grid points and *not* the cursor position, since that will be at some arbitrary position away from the origin.

With respect to the size of objects, gridding has additional complexities. In particular, assuming the top left corner is the origin, should the *width and height* jump to the grid, or the *position* of the bottom left corner. Note that with a grid of 10, the first corner might be at position 0 and the opposite corner at position 10, which gives a width of *11*, and if the outline of the object is 1 pixel wide, then the *inside* of the object will be 9 pixels wide. This also provides for adjacent objects to exactly overlap their outlines, so all the edges look the same size, as opposed to having objects *adjacent*, where the outlines would look to be double-wide (see Figure 10.10(a) and (b)).

10.6.6 Snapping

Whereas gridding causes objects to jump to fixed points in the window, *snapping* refers to having objects jump to positions or sizes with respect to *other objects* [Bier and Stone 1986]. For example, Figure 10.11 shows the endpoint of a line snapping to the point of the triangle when the cursor is close enough. This makes it easier for

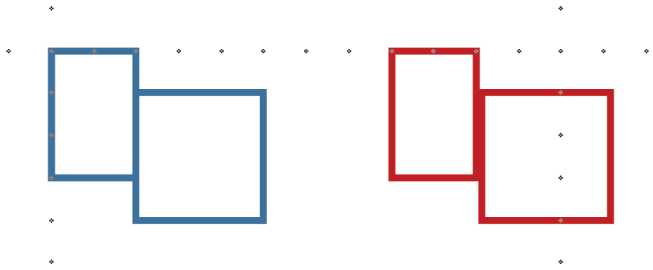


Figure 10.10 Gridding causes the edges of objects to exactly *overlap* (a) and *not* be adjacent (b). Sources: Screen captures from PowerPoint 2016 (v. 16.0), used with permission from Microsoft.

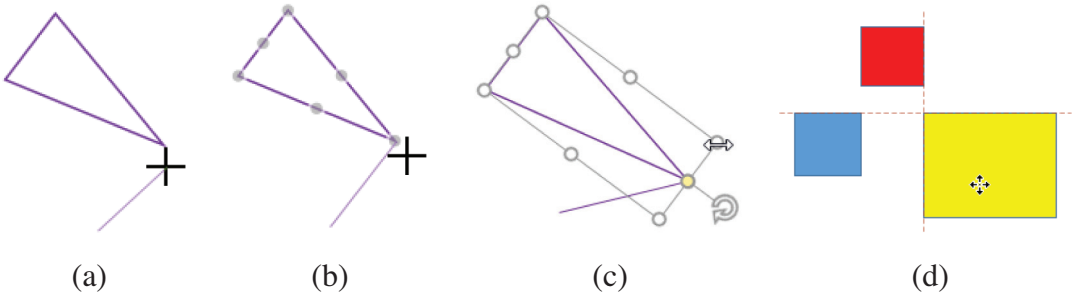


Figure 10.11 Whereas normally (a) the end of the line precisely follows the cursor, shown by the "+", (b) when the line is close to an object, it *snaps* away from the cursor to the point on the object, and the snapping points on the object are highlighted. (c) After snapping, the connection is retained as a *constraint* if either object moves or changes size. (d) When dragging the yellow square, it snaps to "smart guides" (dotted red lines) based on the other objects in the vicinity. Sources: Screen captures from PowerPoint 2016 (v. 16.0), used with permission from Microsoft.

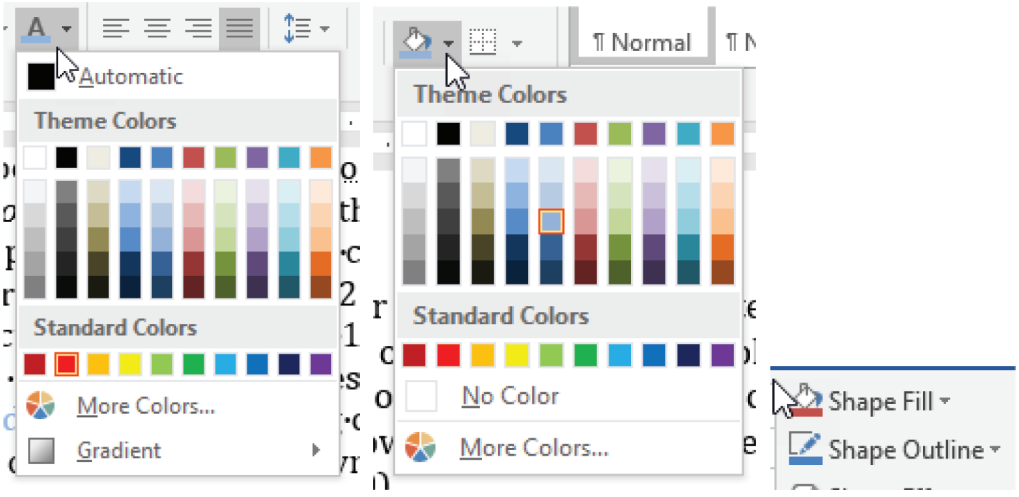
the user to get objects precisely drawn, but it does make it impossible to have the line end *nearby*, but not attached to the triangle, without turning the snapping off first. When both gridding and snapping are turned on, then interaction techniques will normally prioritize snapping over gridding. In some systems, when objects are snapped together, this relationship is maintained if other properties of objects are edited (like when the size of the triangle is increased in Figure 10.11(c), the line will stay attached), thus forming a kind of *constraint*.

Another form of snapping is *aligning* objects. Conventionally, drawing programs will provide a command that aligns one edge or the center of a collection of objects with each other or with respect to the surrounding window. An issue with

this is which object is used as the reference to which the other objects move. Modern systems often have *smart guides* where guidelines are shown as the user moves around or resizes an object based on alignments with the edges of other objects in the scene (Figure 10.11(d)). This interaction technique dates back to the Gargoyle editor from 1985 that provided both manually specified and automatic ("smart") guidelines that appeared based on the drawn objects [Bier and Stone 1986].

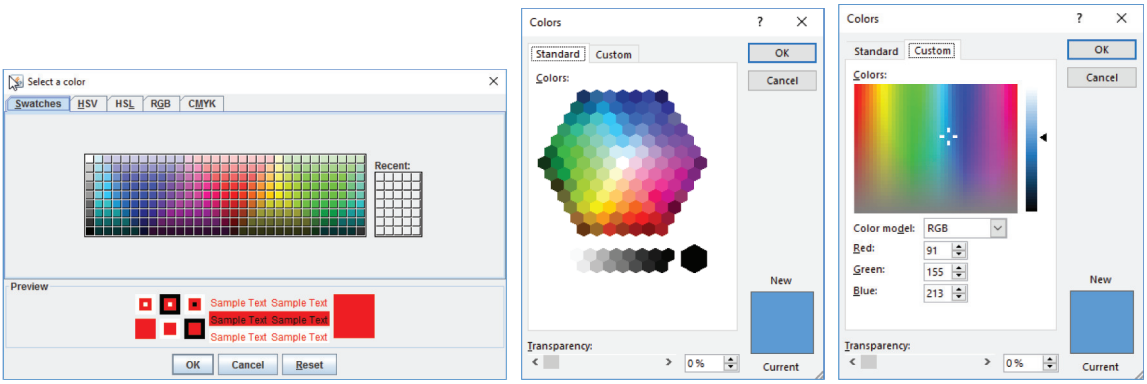
10.7 Specifying Colors

Typically, the user gets to choose the *color* of text or graphical objects. Often objects actually have *two* colors—for text, it is the color of the text and the color of the background, and for graphical objects, it is the color of the *outline* and the color of the insides (or *fill* color). For example, Figure 10.12 shows the widgets for selecting text and graphics colors in Microsoft Word 2016 (v. 16.0). There are special-purpose widgets in the ribbon that are basically combo boxes—clicking on the left side uses the current color shown in the widget, whereas clicking on the down-arrow pops up the dialog to change the current color. The color shown in the icon in Word does *not* change to show the current color of the selected object. However, the pull-down dialog *does* show the current color by surrounding the current object's color with an extra border. For example, in Figure 10.12(a), the current default color is blue, shown by the color under the "A" icon at the top, so clicking on the "A" while text



(a) (b) (c)

Figure 10.12 Widgets and dropdown dialogs for selecting the text (a) color and (b) background color, and the widgets for a graphical object's (c) fill and outline. Sources: Screen captures from Word 2016 (v. 16.0), used with permission from Microsoft.



(a) (b) (c)

Figure 10.13 Color chooser widgets. (a) Built-in for Java Platform SE 7. The (b) standard and (c) custom "More Colors" dialogs from Word and PowerPoint 2016 (v. 16.0). Sources: Screen captures, used with permission from Microsoft.

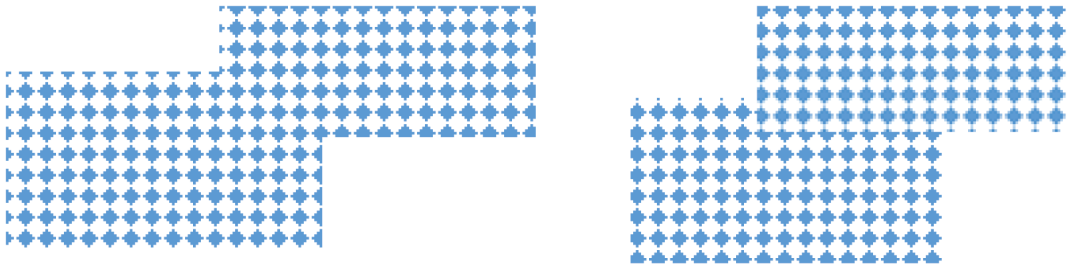
is selected will turn it blue, but the currently selected text is red, as shown by the outline of the red square in the "Standard Colors" section of the pulldown dialog.

There are a wide variety of dialogs available for selecting a color, often called a *color chooser* or *color picker* dialog. For example, Java supplies a built-in `JColorChooser` class (Figure 10.13(a)), and there are many standard third-party color choosers available for virtually every platform and language as well. They usually will have an option to choose one of the standard colors, as in Figures 10.12(a) and (b) and 10.13(a) and (b). There is usually also an option to create a custom color, often using sliders or typing in three numbers.

While color spaces and the physics of color are beyond the scope of this book (see a textbook about graphics [Newman and Sproull 1973, Foley et al. 1990]), suffice it to say that most hardware and low-level tools define colors using a combination of red-green-blue (RGB). This model is harder to use when trying to pick some colors (where is *brown*, for example?), so many systems provide users with other ways to select colors. In some systems, user must still convert the desired colors into an RGB value—for example, HTML defines colors using RGB values from 0 to 255 and encodes these as two hexadecimal digits, such as `#5b9ad5=rgb(91,154,213)` for the color shown in Figure 10.13(c). Other popular color models include CMYK, HSV, and HSL.

10.8 Specifying Other Graphical Properties

There are many other properties of graphical objects that may be specified and manipulated depending on the graphics program. However, most of these would



(a) (b)

Figure 10.14 (a) A texture pattern can be anchored to the window so multiple objects match up, or (b) anchored to objects so it stays consistent when the object is moved, but then there may be seams between objects. Sources: Screen captures.

be controlled with conventional widgets such as menus, sliders, and text boxes that are placed outside of the graphics area (e.g., in palettes, the ribbon or menu bars). Some examples include line thickness, line dashes, transparency, shadows, textures, animations (including blinking), changes when the user interacts with the object (like a different color for when the mouse is hovering over the object), and so on.

A small issue with filling objects with a texture is to control the origin of the pattern. This particularly makes a difference when multiple objects overlap. Typically, the pattern will be anchored by the origin of the picture, so there is no seam in the pattern for overlapping objects (Figure 10.14(a)). However, this means that the pattern differs depending on where the object is. The alternative is for the texture to be anchored to the object's origin, which means that the pattern will always look the same as the object moves, but then the pattern will not line up with adjacent objects, and a seam may be visible (Figure 10.14(b)).

10.9 Specifying Animations and Behaviors

The ways that users can specify that an object changes over time (*animation*) or as a result of a user action (a *behavior*) usually involve writing code (which may be as simple as specifying the change in a parameter in a CSS `hover` attribute for the web or may require general purpose programming) or just using menus or other widgets as discussed above. However, two interaction techniques used for this allow the user to show *before-and-after* versions and *motion paths*. In a before-and-after interaction, which is often classified as *programming by example* (PBE), also called *programming by demonstration* (PBD) [Cypher et al. 1993], the user shows what the object should look like in one state, tells the system to remember it, and then edits the object for how it should look in the other state, which is also remembered

[Myers 1990c]. The user also specifies or demonstrates the trigger or action that should cause the change. This change may also be specified to happen through time rather than instantaneously, so it animates from one value to the other. When the change is to a position, then the user may be able to draw or demonstrate a path on the screen through which it should animate smoothly.

10.10 Selection in Spreadsheets

The previous sections of this chapter were mostly about selecting and manipulating graphical items, but selection is fundamental to all computer interfaces. This section summarizes how selection works in spreadsheets (tables), and the next few sections cover selections in a few other contexts.

Spreadsheets (Section 2.5.1.7 and Chapter 26) are a form of table and provide a grid of cells, so the user needs a way to select one or more cells and one or more rows and/or columns. Originally, the *contents* of spreadsheet cells could only be edited in the separate formula bar at the top of the window that always shows the contents of the selected cell, but current spreadsheets allow the user to double-click on cells to cause the text cursor to appear directly in the cell for text editing in place. If a cell is selected, then the arrow keys usually move the selection to the next cell in that direction (different from what arrow keys do in graphical editors where they *move* the selected object rather than change which object is selected). However, if the text cursor is showing in the formula bar or in the cell itself, then the arrow keys move the text cursor instead (similar to the distinction between nudging and moving the text cursor discussed in Section 10.6.3).

If the user clicks on a cell, it becomes selected. Pressing and holding starts the selection of all the cells passed through by the cursor, as long as the mouse button is held down. Shift-clicking on a different cell causes the *rectangle of cells* from the original to the Shift-clicked cell to be selected, and pressing and holding the mouse button continues to change the selection. The user can switch sides to select cells that are to the upper left of the original cell as well. Note that this is quite different than what Shift-clicking does for graphical objects and is more akin to the behavior for selecting text. Originally, spreadsheets only allowed one rectangle of cells to be selected, but modern spreadsheets allow a new rectangle of selected cells to be started by Ctrl-clicking on them—and Ctrl-dragging can be used for another rectangle of cells. Shift-clicking after Ctrl-clicking expands the most recent selection. Ctrl-Shift starts over with a single cell. Clicking on a row or column header selects that entire row or column, and this can be augmented using Shift and Ctrl pressing as well. Modern spreadsheets allow multiple cells to be *merged* so that the entire merged cell is operated on together, which means that the system must remember

which column or row it was on as it goes through the merged cell so that it can select the correct cell as it moves past it.

10.11 Selections in Linear and Hierarchical Lists

Since the original display screens, computers have always had lists and hierarchies with selections. For example, menus on early computers were just lists of words that supported selections using arrow keys and/or pointing devices. Here, we are primarily discussing list and hierarchy *widgets*, such as might be used to select one or more files or items to be purchased.

When *multiple items* can be selected in a list, the issues around selection become more interesting. With checkboxes (Section 7.2), clicking on an item causes it to toggle whether it is selected (and does not affect whether other checkboxes are selected), whereas in all other contexts, clicking on an item causes it to be the *only* item selected and everything else to be deselected. Pressing and dragging in a list widget usually *moves* the item that was pressed on, typically starting a drag-and-drop operation (which is similar to pressing and dragging in graphics, but different from pressing and dragging in a regular menu, which changes which item is selected as long as the button is held down). In lists, as in spreadsheets and text editing (but *not* like in graphics), Shift-clicking extends the selection from the original point to the item Shift-clicked on. To toggle the selected item and get non-contiguous items selected, Ctrl-click is used. Sometimes, one can select a block of items by clicking in an empty part of the background and drawing out a bounding rectangle, to select (or toggle the selection) all the list items inside the rectangle, like for graphical items.

Most toolkits also supply a widget that provides a *hierarchical view*, also called a *tree view* or *outline view*, where items that have one or more components have an icon that is usually a +/- sign or a sideways or downward pointing triangle (like an arrow) to display or hide the contents of that item (see Figure 10.15). An example is the Windows File Explorer view where folders can contain files and other folders (Figure 10.15(b)). The containers are called the *parent* items and the items in them are called the *children* items. As to the behavior of these widgets, usually clicking on the icon will open and close the view of the children. Clicking on the name of the parent will sometimes open its children and other times just displays the contents of the parent in another pane. Sometimes, users are allowed to select multiple items, in which case hierarchy views work like linear list views.

10.12 Selection in Consumer Electronics

Consumer electronic devices that have screens often provide data that can be selected. Sometimes these are just menus, as described in Chapter 7. An example

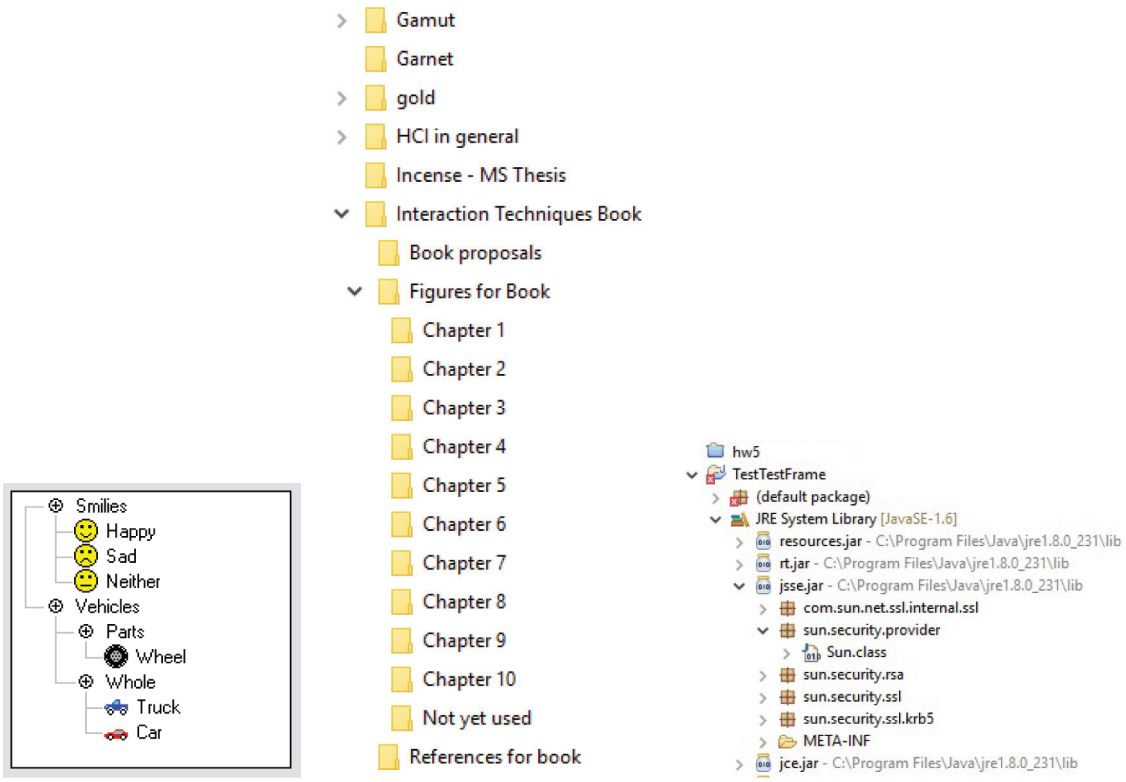


Figure 10.15 Hierarchy views. (a) Generic. (b) File Explorer in Windows 10. (c) Eclipse editor for Java version Neon.3 Release (4.6.3). Sources: (a) Wikimedia Commons, Public Domain. (b) and (c) Screen captures.

of selection that I would *not* classify as a menu is the display of television schedules on a screen, which are sometimes called an *electronic programming guide* or *interactive programming guide* (see Figure 10.16). Here the selection is usually moved using arrow keys on a remote (Figure 4.4(b)). Note that some of the items in the schedule are more than one cell wide—so movement up and down across that cell needs to keep track of what cell to move to. For example, consider the selection starting at channel 2 at 8:30 in Figure 10.16 and moving down—most systems will select the entire program for channel 3, spanning from 8:00 to 9:00, but if the user then moves back up, it should select only the 8:30 cell for channel 2. Similarly, if channel 2 at 8:00 is selected, and the user moves down 1 and then to the right 1, typically the 9:00 cell will be selected, and then moving up should select the 9:00 cell for channel 2 as well. This is similar to how spreadsheets handle moving through merged cells (Section 10.10).

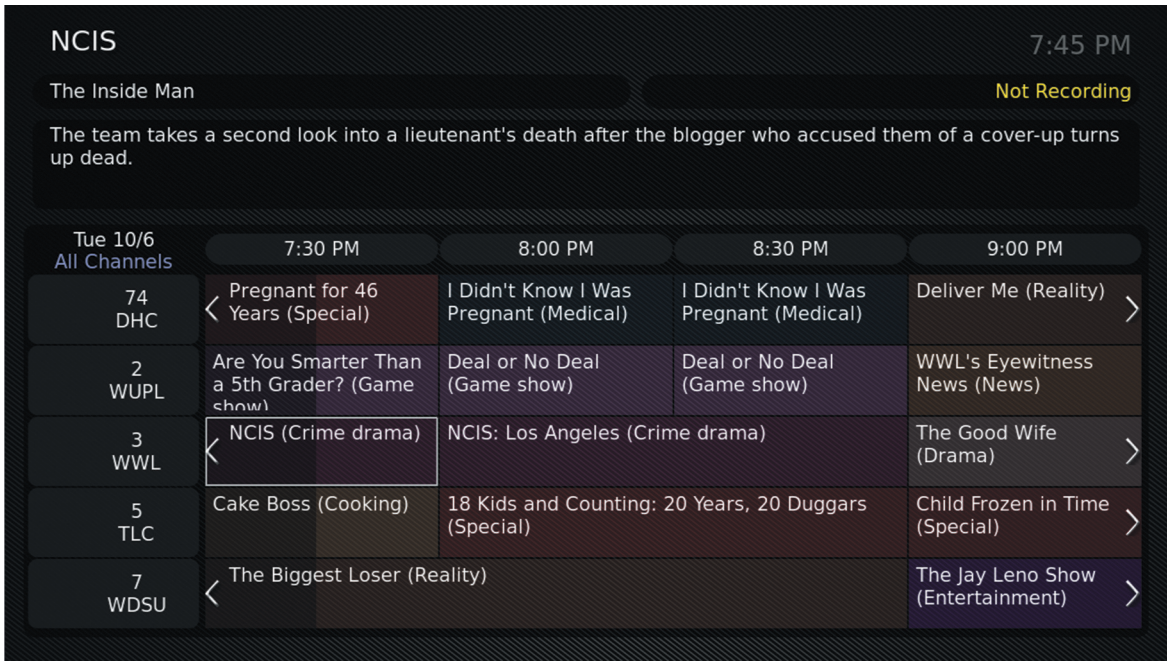


Figure 10.16 Electronic programming guide as a 2D grid. Source: By Mythbuntu team, GPL, https://commons.wikimedia.org/w/index.php?curid=12515338.

10.13 Brief History of Graphical Editing

This section summarizes the key aspects for how various historical systems have supporting graphical selection and editing, presented in approximately chronological order.

- Sketchpad, 1963 (Section 2.4.1)
  - Drawing was *prefix*, using buttons and switches to define the operation, and then the user provided the points using the lightpen. Some operations were postfix—user would point to one object with the lightpen and then specify something about it with the switches, like "Make a line be vertical."
  - No "selection set"—just what object was currently being pointed at, where the cursor was shown.
  - Objects could be edited by pointing to points, operating a physical switch, and then dragging out to show the new location for the point.
  - Supported grouping of graphical objects, subpictures, and inheritance of properties from a prototype object with the ability to edit the prototype.

- Xerox Draw, 1975 (Section 2.4.3.3)
    - Drawing was *prefix*, but used postfix for editing graphics.
    - Operated on the selected object or created a copy of the selected object.
    - Separate commands for translation and scaling, using affine transformations, with either 2-point, 4-point, or 6-point transformations.

      Translate—select object, then switch to translation tool and click twice (2 points) to show the translation vector (where the vector was not necessarily on the object).

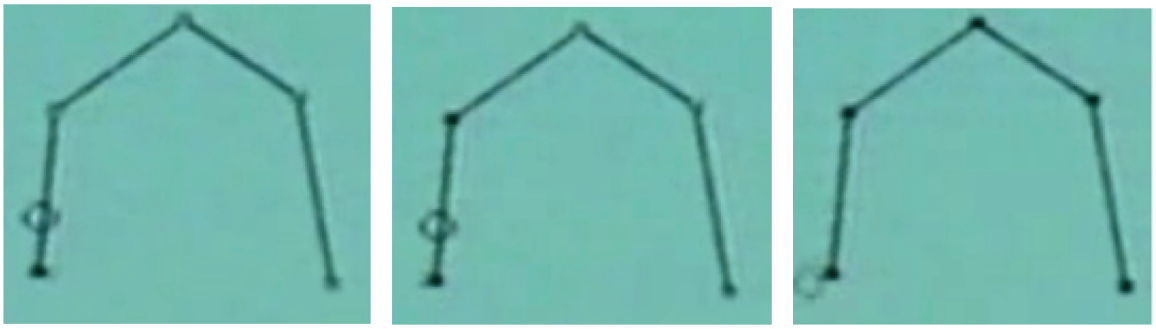
      4-point—transformed first vector into the second vector and did the same to the selected graphics.

      6-point—full affine transformation of the selected graphics.
    - No interim feedback while drawing or editing.
    - Single select anywhere on object with any mouse button.
    - Area select—needed to press mouse button down in an unoccupied area, drag out a rectangle, and selected everything that was totally inside the bounded area.
    - Selection was shown with single square on the object.
- Xerox Star, 1981 (Section 2.4.4)
    - Used function keys on the keyboard for Move, Copy, and so on.
    - Select object, hit and release move key (cursor shows the mode), and click where object should go.

      "Stretch" was a menu item at top of window, or a keyboard key, then press and drag to get continuous feedback.

      "Magnify"—change size but keep same proportions.
    - Feedback was object itself changing for resizing operations, no feedback for moving until final position.
    - Click on objects to select them, and the selection was shown with many handles.
    - Right click on objects to add to the selection.
    - Right mouse button drag to get a rubber band rectangle for selection in a region.

- Macintosh, 1984 (Section 2.4.7)
    - Selected files were shown in reverse video.
    - Selected graphics were shown with boxes at four corners.
    - Click to select, Shift-click to toggle in selection.
    - Area select—press down in an unoccupied area and then drag to completely enclose objects to be selected.
    - Holding down the Shift key toggled the selection, or when dragging with the Shift key and the mouse button, it toggled objects in the selected set.
    - MacPaint provided the *lasso tool*, which shrinks down to select the pixels, ignoring surrounding white areas.
    - Move objects by pressing and dragging anywhere inside the selected object that is not over a handle.
    - Resize by pressing on a handle.
    - Interim feedback used a dotted bounding box (not the object itself).
- Gargoyle, 1985, [Pier et al. 1988] (Section 2.4.3.3)
    - Gargoyle was an experimental graphics editor from Xerox PARC that provided novel ways to create and edit graphics, particularly curves and sequences of connected lines. It also was one of the first systems to support snapping [Bier and Stone 1986].
    - Operations on objects included translate, rotate, and skew.
    - It provided a separate *anchor* and *cursor* (shown by a caret), making it easier to define the object to be manipulated, and reference points (e.g., to draw from, rotate around, or the vector to stretch).
    - The caret snapped to objects to provide precise rotation and scaling.
    - Feedback used the object itself moving or changing (the interim feedback was the object itself moving).
    - Single click on an object selected the nearest vertex; multiple clicks went up to selecting the whole segment, then the whole object, and so forth (Figure 10.17). Dragging out a rectangular area would select everything inside [Myers 1990b at 1:33:48].
    - Small black squares showed what was selected, with open squares to show the rest of that object, and thin rectangles surrounded clusters



(a) (b) (c)

Figure 10.17 Gargoyle editor from Xerox PARC, selecting (a) the lower left endpoint with a single click, then (b) clicking again selects the segment, and (c) clicking a third time selects the whole object. Sources: Myers [1990b] at 1:33:48–1:33:50.

(a) (b)

Figure 10.18 Alternate Reality Kit showing the cursor as a hand and a slider (a) before and (b) after the slider is selected and picked up. Sources: Myers [1990b] at 1:34:16–1:34:23.

of selected objects. The caret also jumped away from the mouse cursor to show when the point would be snapped to that point.

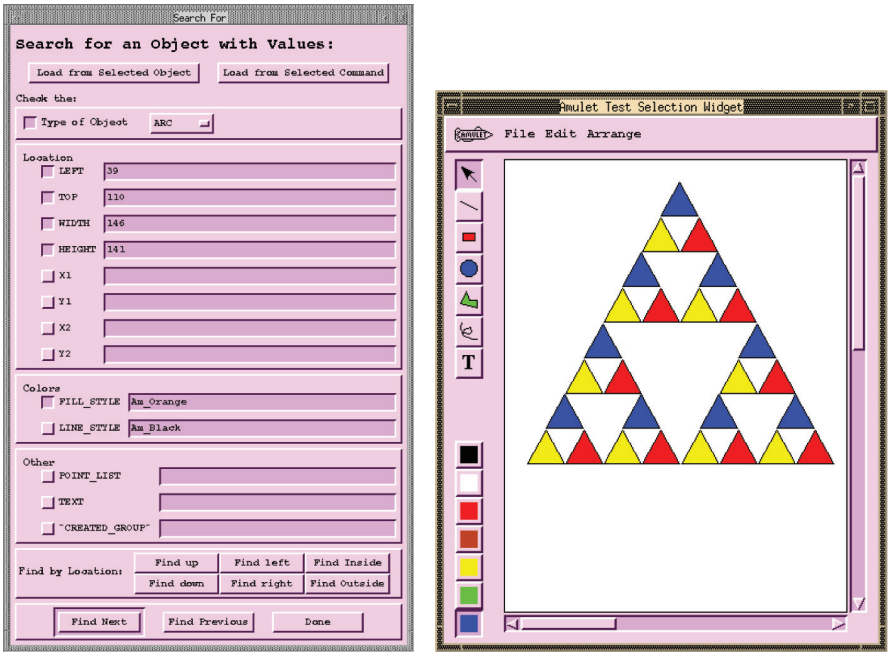
- The Alternate Reality Kit, 1985 (Section 2.4.10) [Smith 1986, 1987]
  - The Alternate Reality Kit took a radical *skeuomorphic* approach (making graphics look and behave like real-world objects) so the feedback that an object was selected was that it appeared to rise off of the surface of the screen in 3D (by shifting slightly and displaying a shadow—see Figure 10.18), and the cursor, which was represented as a hand, appeared to close around the object as if it was being carried.
  - The object in the hand would then follow the mouse (move) and could be dropped by releasing the mouse button, or set in motion (animating) by throwing it while releasing.

10.14 Research in Selection and Editing Objects

Although some of these interaction techniques have been pretty consistent since SketchPad in the 1960s, Xerox PARC systems in the 1970s, or the Apple Lisa/Macintosh in the 1980s, other aspects have been continually researched.

One interesting avenue has been *search*. Although textual search has always been provided in text editors, *graphical search* is generally not provided even though David Kurlander in 1988 showed that graphical search and replace could be supported and would provide many benefits [Kurlander and Bier 1988]. Ten years later, I built on this result in my Topaz editor [Myers 1998b], which provided a dialog box to define the properties of object to be searched for (see Figure 10.19) and, if desired, what value should be replaced. Topaz also provided other features, including the ability to change which object is selected using keyboard keys, recording a sequence of operations for later replay (discussed in Section 15.13), and generalizing the script so it could be used in different contexts [Myers 1998b].

Another dimension of research involves multiple people editing the graphics at the same time. As with multi-user text editing (Section 8.6.4), it is important that



(a) (b)

Figure 10.19 The Topaz research graphical editor showing (a) the search dialog and (b) the result of replacing a triangle with three triangles, applied recursively multiple times. Sources: Myers [1998b].

each person be able to see where the other person is editing, usually by having a distinct cursor for each user. A typical way to distinguish them is to give each user a different color, such as in the MMM system (Multi-device Multi-user Multi-editor) that supported multiple mice connected to the same monitor [Bier and Freeman 1991]. In my PebblesDraw system [Myers et al. 1998], we decided to give each user their own *shape* instead, and to use that shape also for the user's selection handles (instead of just a square or a circle), so each user can have their own set of selected objects and their own current color for drawing (see Figure 4.18).

Research systems have investigated how to make it easier to move information across the same or different computers. Jun Rekimoto in 1997 investigated *Pick-and-Drop*, where a stylus that can be identified by an ID is used to "carry" a digital object from one computer to another just by tapping on the screen of each in turn [Rekimoto 1997]. We showed it was possible to just use copy-and-paste across multiple computers (even of different brands) in 1999 [Miller and Myers 1999], which finally became available across Apple devices in 2016, which Apple calls *Universal Clipboard*. Other research tries to improve drag-and-drop in various ways. For example, Drag-and-Pop (mentioned in Section 4.15) detects the direction the cursor is moving and temporarily moves the targets that can be reached in that direction to be closer to the cursor so that they can be selected more easily [Baudisch et al. 2003]. Fold-and-drop makes it easier to drop objects into windows that are fully covered by other windows by "folding" over the corners of the obscuring window [Dragicevic 2004].

Research into new ways of *drawing* complex shapes has also continued. For example, the team in Toronto that was originally Alias then Alias|Wavefront, and now is part of Autodesk Research, has invented a large number of interaction techniques for selecting and controlling drawing tools and their parameters, both for mouse-based and multi-finger drawing. For example, a 1999 paper investigated more precise control over freehand drawn curves by using one hand to control the curvature while the other hand defines the shape [Balakrishnan et al. 1999]. Since conventional palettes can be awkward to use with fingers due to the potentially tiny size of the targets, their *object-oriented drawing* system from 2016 researched more effective ways to use one and multiple fingers on a touch tablet. It has *attribute objects* (see Figure 10.20) that encapsulate parameters to the drawings into manipulable "cards" that can be adjusted using multiple fingers [Xia et al. 2016]. Many of these innovations have been integrated into their graphical editor products such as their high-end professional 3D tool called Maya.

There is also significant research into ways to make it easier to *program* graphical editors and selections, which will be covered in Chapter 20. For example, my Garnet [Myers et al. 1990] and Amulet [Myers et al. 1997a] toolkits included built-in

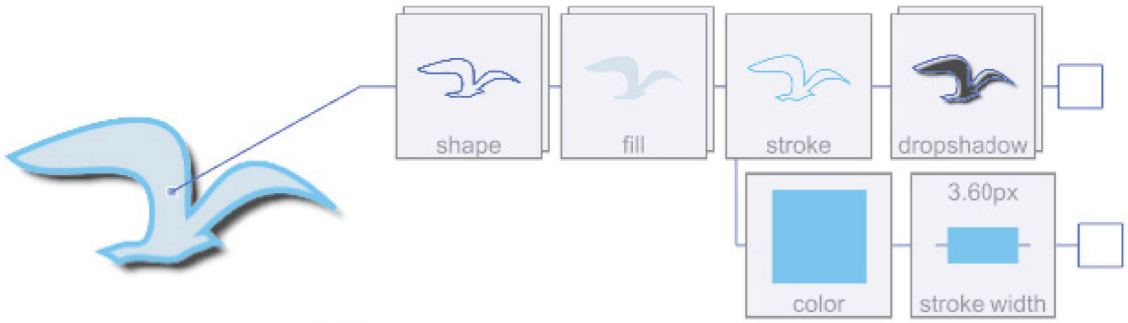


Figure 10.20 The properties of the drawing of the seagull are represented in the cards on the right, which can be easily manipulated using one or more fingers. Source: Xia et al. [2016].

selection-handle widgets that handled selecting, moving- and growing objects automatically, which requires significant programming in most other toolkits.

10.15 Summary of Considerations for Selecting and Creating Objects

Most of the issues that an interaction designer would need to consider for creating, selecting, and manipulating objects have been covered in this chapter, so this section just summarizes them.

- Whether this is a painting (pixel-oriented) or drawing (object-oriented) editor.
- The syntax of the operations (e.g., is the selection first or tool first?). Selecting the appropriate syntax may depend on whether selecting the objects or the operation is more error prone.
- Support for grouping.
- How objects are created.
- Whether selection of multiple objects is supported. If so, the specific interaction techniques to do that.
    - Whether Shift-click selects objects in a range (like in text and lists) or toggles the selection (like for graphics).
- Support for selecting outline and thin objects.
- Support for selecting objects that are inside an area defined by the user.
    - To select objects, do they need to be fully inside the area or can they just overlap the area?

    - Shape of the bounding area—just a rectangle or more complex shapes (like for a lasso).

- How the user can make there be nothing selected.
- Can objects *inside* a group be individually selected and operated on, versus just selecting and operating on the group itself?
- How the selection is shown to the user.
- What coordinate system will be used.
- Whether creation and editing operations can be aborted.
- What feedback is shown for editing operations, especially the *interim feedback* while the operation is in progress.
- For objects that are defined by a bounding box, whether the user can move to the upper left (*flip if change sides*), and if so, whether that turns the object upside down or mirrored.
- Whether objects have a minimum size.
- Whether zero-size objects are allowed, and if so, how they can be selected and edited. If not, how they are prevented.
- How objects resize, especially when grouped.
- Whether thick outlines extend to both sides of the coordinates or just to the inside.
- Specialized editing for complex shapes (like polygons).
- Whether constrained resizing (like staying at the same aspect ratio), gridding, and snapping are supported.
    - If so, what origin will be used.
    - Interaction techniques for defining the grid and snapping.
- Specifying other graphical properties such as color, line styles, and so on.
- For other applications (spreadsheets, lists, consumer electronics, etc.), how the user selects items.
    - How the selection changes with various size objects.

10.16 Chapter Conclusion

Although graphical editing is quite specialized, selection is a fundamental operation for many different kinds of applications. A key property of direct manipulation interfaces is to directly identify the objects of interest by pointing at them. For

conversational interfaces, however, users identify the objects of interest and the operations by naming or describing them, and there is often no feedback for which objects will be operated on. For the future, combining direct and indirect selection of objects would be the most flexible approach with the best feedback for the user.

CHAPTER 11

Desktop Metaphors, Window Managers, and Icons

11.1 Introduction

Since the early days of computing, people have envisioned being able to see more than one thing at a time on a computer. As summarized in Section 2.5.1.4, the idea of separating different tasks into different windows, or areas on the screen, dates back to at least 1968 and has been the established way to organize screens since the 1980s. This chapter starts with definitions and a history of windows, and then discusses the many interaction techniques to manipulate the windows and how having multiple windows imposes restrictions on the interaction techniques of applications. In the last 35+ years, there have been a few innovations, but basically, the interaction techniques by which users interact with windows have remained fairly consistent, as shown in this chapter.

11.2 Definitions

First, here are some more formal definitions of the key terms in this chapter. Other important terms are discussed in the following sections.

- *Window manager*—I have defined *window manager* as "a software package that helps the user monitor and control different contexts by separating them physically onto different parts of one or more display screens" [Myers 1988].
- *Window*—Therefore, a *window* is an area of the screen (typically a rectangle) through which the user interacts with one or more processes.

- *Window system*—X Windows for Unix (Section 2.4.12) provided different user-selectable user interfaces for windows on top of the same low-level application programming interface (API), so it called the user interface the *window manager* and the low-level parts the *window system* or *windowing system*. However, this distinction did not catch on, so I will use the terms window manager, window system, and windowing system interchangeably.
- *Imaging model*—All window systems clip the output drawing to the boundaries of the appropriate window, and so all graphics must go through the window system. Therefore, *how the graphics and text are drawn on the screen* is called the *imaging model* of the window system. The original systems only supported black and white drawing and simple graphics, whereas current systems support sophisticated two-dimensional and three-dimensional (3D) rendering and video (see Section 20.12.5).
- *Pane*—When areas of the screen are *divided into subareas such that they do not overlap*, and are adjacent, these are usually called *panes*.
- *Frame*, *viewport*, *canvas*, or *document*: Various window managers may have different names for the *inside* part of the window. The terms *frame*, *viewport*, *canvas*, and *document* have all been used to mean either the window itself or the inside of the window where the real content is (not counting the window title and border areas). These terms may be only visible to programmers using the libraries to output to the window, or they may appear in user-facing documentation. Sometimes, these terms have other meanings in other contexts; for example, *frame* and *canvas* have special meanings in HTML (a *frame* is an old way (pre-`<div>`) to divide an HTML page into different regions, and a *canvas* is an area to paint pixel-oriented graphics).
- *Desktop*: As explained in more detail in Section 2.4.4, the desktop is the full screen *behind* the windows, where the icons and other graphics are displayed. The term comes from the *desktop metaphor*, since the screen is supposed to simulate the top of a desk, with papers, folders (directories), and (for the Xerox Star, Section 2.4.4) an inbox, outbox, and printer, and for all other modern systems, a trashcan or recycle bin. It is common to say that the windows are arranged *on* the desktop, meaning on the screen. Often, the user can select a pattern or picture to display as the background of the desktop.

11.3 A Brief History of Window Managers

The earliest windows managers supported *tiled* windows, which means that the windows are side-by-side and cannot overlap (see discussion in Section 11.5).

Multiple tiled windows were demonstrated in Engelbart's NLS system in 1968. Early research at Stanford on systems like COPILOT (1974) [Swinehart 1974] and at MIT with the Emacs text editor (1974) [Stallman 1979] also demonstrated tiled windows, and they were used in the original VisiCalc around 1977 (see Section 2.4.4). Alan Kay proposed the idea of overlapping windows in his 1969 University of Utah Ph.D. thesis [Kay 1969], and they first appeared in 1974 in his Smalltalk system (Section 2.4.3.1), developed at Xerox PARC, and soon after in the Interlisp system [Teitelman 1979], also at Xerox PARC. Some of the first commercial uses of windows were on Lisp Machines Inc. (LMI) and Symbolics Lisp Machines (1979), which grew out of MIT AI Lab projects (see Section 2.2.2). The Cedar Window Manager from Xerox PARC was the first major tiled window manager (1982) [Swinehart et al. 1986], soon followed by the Andrew window manager (called "wm") [Palay et al. 1988] from Carnegie Mellon University's Information Technology Center (ITC) (1983, funded by IBM). The main commercial systems popularizing windows were the Xerox Star (1981) (Section 2.4.4), the Apple Lisa (1983), and most importantly, the Apple Macintosh (1984) (Section 2.4.7).

The first version of the Xerox Star used two columns of tiled windows, which was replaced with overlapping windows in the second version from around 1985, called *Viewpoint*. Similarly, Microsoft Windows version 1.0 used tiled windows, primarily because Microsoft hired Scott A. McGregor from Xerox PARC to lead the Windows project (see discussion in Section 31.3). McGregor was one of the key designers and implementers of the Cedar environment (Figure 11.3), which was tiled. However, due to customer demand, Windows moved to overlapping windows in version 2.0 from 1987, but it was Windows 3.1 from 1992 that really started to be widely used (see Section 2.4.8). Meanwhile, IBM in collaboration with Microsoft, and IBM alone after 1990, created a competing window manager called Presentation Manager for a PC operating system called *OS/2*, first released in 1988. The interaction techniques in Presentation Manager were very similar to Windows 2.0 and later. Microsoft developed numerous versions of Windows, forking into two branches for a while starting in 1993 (32-bit Windows NT vs. regular 16/32-bit consumer Windows). Again, the interaction techniques on these were similar. The next key changes to the UI of Windows came with Windows 95 in 1995, which introduced the Taskbar and related interactions (Section 11.7).

Meanwhile, for the Apple Macintosh line, the interaction techniques stayed fairly constant, as Steve Jobs was forced to leave Apple and formed the NeXT computer company in 1985 (Section 2.4.11). The NeXT window manager (see Figure 2.20) was deliberately made to look different from the Macintosh—it changed the menu bar to be a vertical fixed menu with pullouts and added a dock taskbar on the right. When Jobs rejoined Apple in 1997, he brought many aspects of the NeXT window manager into the Macintosh, including the dock.

As mentioned above, the Lisp Machine workstations were some of the first implementations of window managers, starting around 1979. Other workstation companies (Section 2.2.2) developed their own proprietary window managers, with similar interaction techniques, providing windows with title bars that could be dragged around, menus for other commands, and icons, starting around 1980. However, the programming interfaces were different, causing applications to need to be implemented multiple times, which was onerous. This inspired a push for an open-source common window manager, which eventually resulted in Project Athena at MIT creating a new windowing system, starting about 1984 [Scheifler and Gettys 1986]. A story is that they originally asked the Andrew project if they could use the Andrew window manager (called "wm"), but IBM, who funded the Andrew project, did not agree, so Project Athena used the next letter for their window manager, which was therefore called *X Windows*. Since X ran on multiple platforms, an important goal was to allow a variety of window managers, each with their own interaction techniques, to be used with X. This was successful, and window managers proliferated, with examples such as uwm [Gancarz 1986], TWM, and so on. As discussed in Section 2.4.12, the major hardware vendors like Sun, HP, IBM, and so on wanted a more professional looking window manager and set of widgets, so they formed consortiums, which came out with two competing window managers: OpenLook, based on the Xerox Star look and feel (see Figure 2.21) and supported by IBM, Sun, Xerox and others; and Motif, with a more 3D (skeuomorphic) look (Figure 2.22), supported by HP, DEC, and others. Motif took better advantage of color screens and eventually won out, with OpenLook disappearing around 1993.

11.4 Input Focus

An important aspect of window managers is how the user changes which window is connected to the keyboard and potentially other input devices. This window is termed the *input* (or keyboard) *focus.* Another term I have used is the *listener*, since it is listening to the user's typing [Myers 1988]. Older systems called the focus the *active window* or *current window*, but these are poor terms since in a multiprocessing system many windows can be actively outputting information at the same time. Window managers provide various interaction techniques for selecting and showing which window is the focus.

As discussed in Section 5.9.2, in modern systems, users can scroll any window whether it is covered or not just by moving the pointer cursor over a window. However, most systems have required the window with the keyboard focus to be at the front. This means that the interaction technique that brings a window to the front can *at the same time* change the focus window. For example, in the Macintosh and modern Windows, clicking inside a window that is not the focus both brings it to

the top *and* gives it the keyboard focus. This is called *click-to-type*. In some older window managers, like my Sapphire [Myers 1984] and Unix window managers like SunWindows, just *moving* a cursor into a window caused it to get the keyboard focus, which is called *move to type*, and a separate click on the title line or other action was needed to bring that window to the front.

With click-to-type, the next question is whether the click that causes the window to get focus is *also* sent through to the application. This was always done in some window managers, such as Cedar (see Figure 11.3), so the user had to be careful where to click if it was desired to just give focus to the window. In most systems, however, the press that changes windows is *not* sent to the underlying program, so it will not cause a selection or button press of the window. For Microsoft Windows, however, it often depends on where one presses and what application is running. For example, in Windows 10, pressing (but not releasing) on the title line of a covered window brings it to the top, gives it the focus, *and* starts that window following the mouse (moving the window). Similarly, pressing and holding the mouse button in a non-focus window that is running Adobe Acrobat Pro DC (version 2019.022) in Windows 10 will bring the window to the front, give it the focus, *and* start selecting text in Acrobat, whereas doing the same operation in Microsoft Word 2016 or 365 will *not* start selecting text, which would require a second mouse press. With most text editors, like Microsoft Word and Google Docs, clicking in a non-focus window will give it the focus *and* move the text cursor, so to change focus without moving the text cursor, a good trick is to click in non-text regions like the scrollbar, title, or menu areas.

The Macintosh menu bar at the top of the screen operates on the window that currently has the input focus. In current systems, the user can move the selected menu item in the menu using the keyboard arrow keys, so the menu bar itself actually has the focus, but the system shows a window with focus. Notice that the global menu bar requires using click-to-type since with move-to-type the cursor might pass through other windows on the way from the current window to the top of the screen, thus changing the focus accidentally. A related issue is the Find dialog in Microsoft Office and other systems. This is a top-level, non-modal window that receives the focus so the user can type the characters to find. However, the find operation happens in the *previous* window that had focus, so this must be kept track of.

Another issue is whether the window manager allows there to be *no* window in focus. In the uwm window manager for the X system, for example, when the cursor is outside of all windows (that is, over the background), keyboard typing just disappears. In Sapphire [Myers 1984], keys typed when there was no focus were saved and given to the next window that got the focus. On the other hand, the original

Macintosh did not allow there to be no focus window; if you pressed over the background, nothing happened, and if the focus window was deleted, then the next topmost window was made the focus.

An important presentation issue is how the focus is shown. In the original Lisa and Macintosh, shown in Figures 5.11 and 5.12, the focus window displays its title line and scrollbars, whereas un-focused windows just show a white area. Many other window managers have changed the display of the title and/or the border around the focus window (e.g., see Figure 11.1). Some older window managers only supported output to the front-most focused window, which allowed the *inside* of the un-focused window to be shown differently. For example, the Blit window manager from AT&T in 1983 did not have a title line and filled the non-focus windows with halftoning dots so it looked grayed out [Pike 1983]. A more direct signal of which window has the keyboard focus is where the text cursor is blinking, if text entry is currently enabled in that window.

Many systems provide alternative ways to change the focus window (which often also brings that window to the top). For example, Microsoft Windows introduced Alt-Tab in Windows 3.0 in 1990 as a way to cycle through the windows one by one (and Alt-Shift-Tab to go backward through the same list), bringing each one to the front and giving it the focus. This was picked up by the Macintosh and now works there as well, using Command-Tab and Command-Shift-Tab. There is often a way to view shrunken versions of all the open windows, so the user can click on the desired one. Alt-Tab in Windows now brings up such a view, as does the Windows key and Tab. The Macintosh has a similar view. On an iPhone, the user can see all the running applications by double-clicking on the home button (or using a swipe-up-and-hold gesture on iPhones without a home button) and then tapping on the desired window. Using these techniques also avoids the issue that clicking in a window to give it the focus might cause an operation or move the text cursor.

11.5 Overlapping Windows versus Tiled versus Tabbed versus Single Window

One very important aspect of the presentation of windows on desktop systems is whether they can *overlap* or not. *Overlapping windows*, sometimes called *covered windows*, are a feature that allows a window to be partially or totally on top of another window, as shown in Smalltalk (Section 2.4.3.1), the Apple Lisa (Figure 5.11), Sapphire (Figure 11.1), and Microsoft Windows starting with version 2 (Section 2.4.8). This is also one of the aspects of the desktop metaphor since windows can cover each other like pieces of paper can cover each other on a desk.

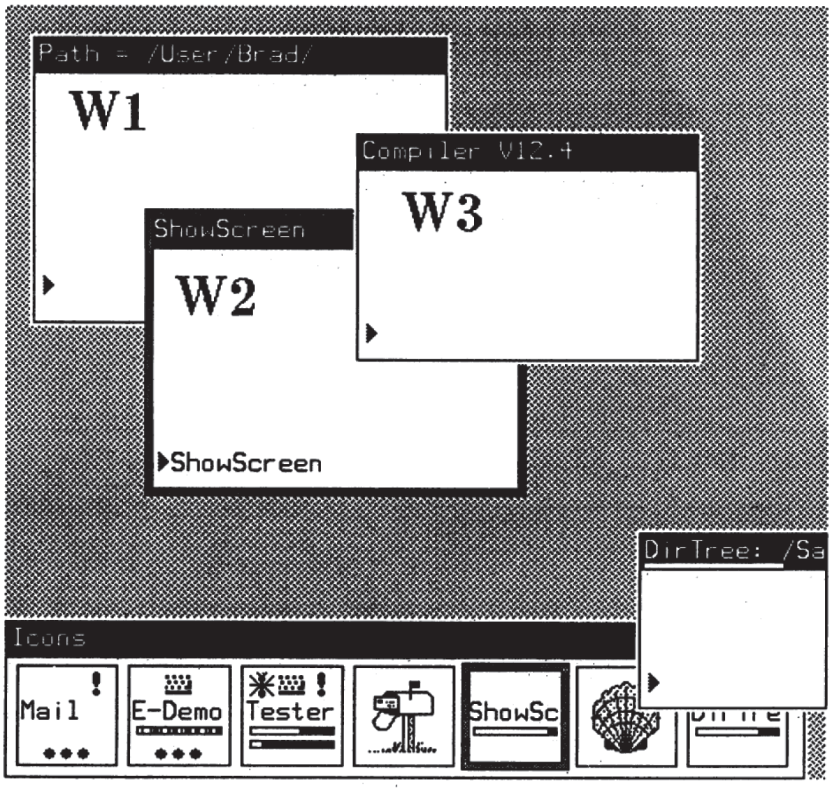


Figure 11.1 A screen from the Sapphire window manager from 1984. The icons are collected into a special window that by default is at the bottom of the screen. W2 has the keyboard focus, as shown by its thicker border, which is also shown on its icon ("ShowSc"). Progress bars (Chapter 6) are shown in the title bar of each window and in most icons (see also Figure 6.3). Source: Myers [1988].

A variation on overlapping windows, provided by Microsoft at least since Windows 3.0 in 1990, is called the *multiple document interface* (MDI). It provides for covered windows *inside* a window that itself is a covered window. For example, Figure 11.2 shows two windows inside the "Program Manager" window that the user can move around and change the one that is on top, and the Program Manager window itself is behind the Reversi and File Manager windows. The advantage of MDI is to keep an application's windows organized within the application, but the disadvantage is reduced flexibility and increased complexity for users in trying to arrange windows as desired, so MDI is rarely seen anymore.

Another alternative is called *tiled* (or *paned*) *windows*, which means that windows are not allowed to cover each other and instead are required to be abutting

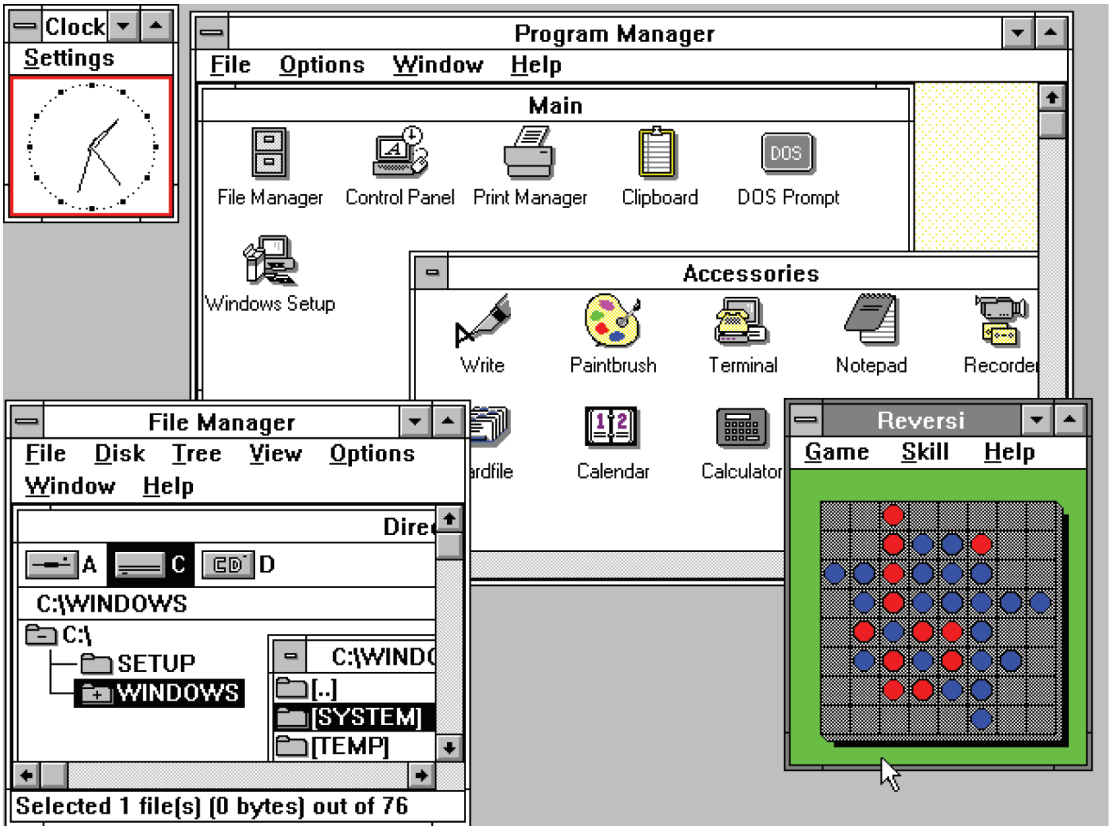


Figure 11.2 The Program Manager window in Windows 3.0 (1990) uses MDI to show the Main and Accessories windows inside of it, each of which can be moved around and changed size, but must stay inside the Program Manager window. Source: Tyomitch, https://en.wikipedia.org/wiki/Windows_3.0#/media/File:Windows_3.0_workspace.png. Used with permission from Microsoft.

side-by-side. Figure 11.3 shows an example of tiled windows. Obviously, a window manager that supports covered windows can also allow them to be side by side, but not vice versa. Therefore, a window manager is classified as covered if it allows windows to overlap. Examples of tiled windows include early editors like Emacs (from 1976 by Richard Stallman, Guy L. Steele, Jr., and Dave Moon) and Bravo (Figure 2.11), window managers like Cedar (Figure 11.3) and Microsoft Windows Version 1.0 (Figure 2.17), and the "split" views in modern tools that show multiple parts of the same document side-by-side or top-to-bottom (available in Microsoft Word and Acrobat).

When windows are tiled, the next decision is whether they must be arranged in fixed columns or whether they can be in arbitrary places on the screen (see

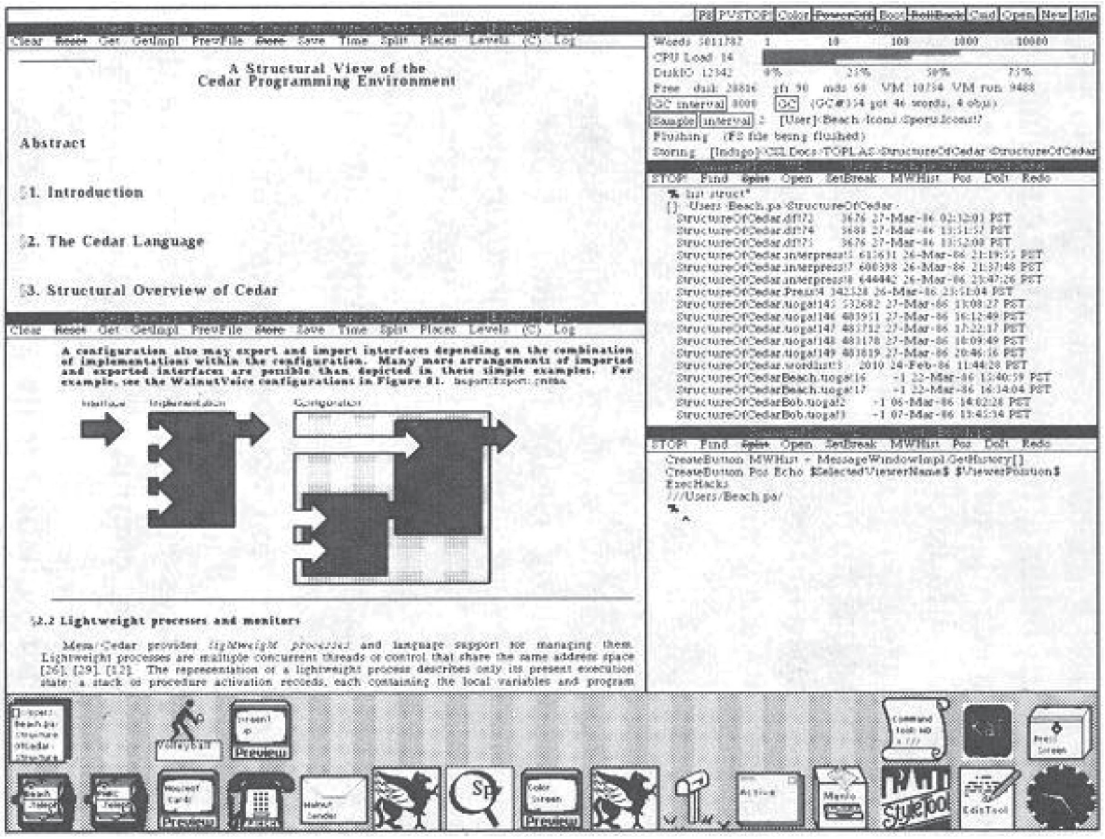


Figure 11.3 A screen from the Cedar environment from Xerox PARC, showing its tiled windows and two rows of icons at the bottom. Most windows are displaying a row of buttons just below the title line. Source: Swinehart et al. [1986].

Figure 11.4). Originally, the Andrew window manager used a constraint system to allow windows to be non-overlapping and anywhere on the screen. The system would adjust the sizes of windows based on the constraints whenever a new window was created or an old window destroyed. Unfortunately, users did not like this for a number of reasons: it took too long for the windows to finish adjusting themselves after a change, windows all over the screen would change size when a new window was added or removed, and the screen layout resulting from one window changing was unpredictable. Therefore, Andrew switched to a much simpler approach, with a user-defined number of columns [Morris et al. 1986]. No system I am aware of uses the arbitrary layout approach. If there are fixed columns, then there can be either a specified number or an arbitrary number. Emacs and Bravo just supported a single column (Emacs later supported multiple columns). Cedar

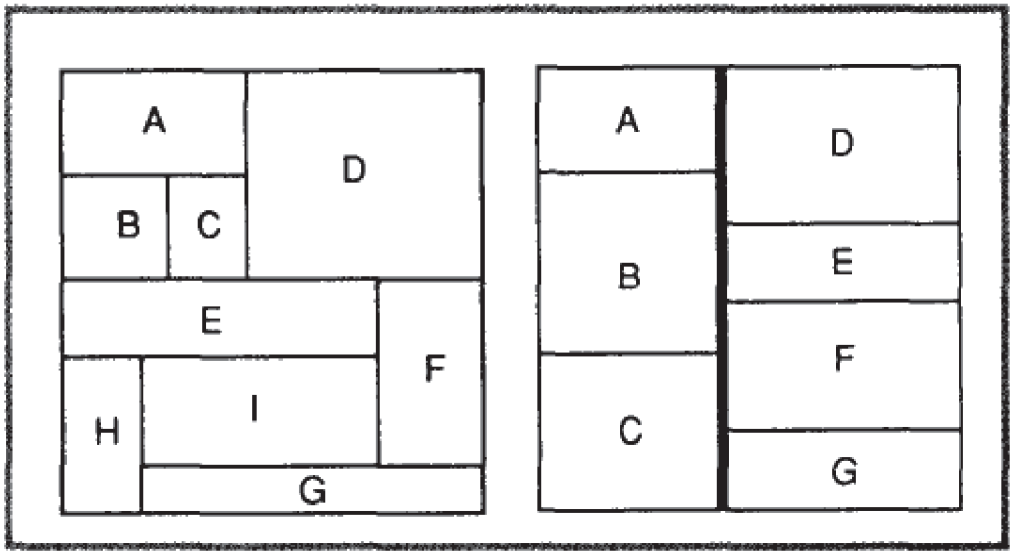


Figure 11.4 Tiled windows can be in arbitrary places (left) or in fixed columns (right). Source: Fig. 6 of Myers [1988].

provided for exactly two columns on a black-and-white display (see Figure 11.3), along with one additional column on an optional color display.

A study in 1986 found that, while users claimed to prefer covered window managers, they spent less time doing window management operations with tiled window managers [Bly and Rosenberg 1986]. In tiling systems, the computer is typically in charge of managing the window placement and size, limiting the user's freedom. In covered window managers, the user must usually manage the windows. Putting the mouse in an arbitrary window is also easier with tiling since all of the windows are always visible. The computer's screen size will also affect whether tiling or covered is preferred. With small screens, such as on the original Macintosh, there is not enough room to use the tiling style. For the graphics system implementer, graphical output primitives are more difficult to provide with covered systems since the output must be clipped to differently shaped regions. On the other hand, with tiled systems, the window manager implementer must provide some sort of automatic screen layout facilities. Some window managers, such as Viewpoint (which is the successor to Star), allowed the user the choice of covered or tiled windows. Another possibility is for a window manager that uses covered windows to provide automatic layout. This is harder than with tiled windows, however, because it is less clear where windows should be placed—often modern systems just lay them out in a grid.

The third way to organize windows is using *tabs* as the title of each window, also called a *tabbed document interface* (TDI), and lining the tabs up in one or more rows or columns. Users can then only see one window at a time, with each window having essentially the same size, but it is easy to switch among the windows using the tabs (assuming there are not too many). (Note that the Smalltalk window system, the first covered window system from 1976, had window titles looking like notebook tabs (see Figure 2.10) but did not organize the windows as overlapping tabs, so it does not count as a TDI.) WordVision, a DOS word processor from 1982, provided file folder tabs to organize the different document files [Seymour 1994]. HyperCard (1987) provided tabs to navigate to different pages (Section 2.4.9). The first use of tabs for windows of different content was probably on notebook computers like PenPoint from around 1991 (see Section 3.5), where the tabs were lined up in a column on the right side of the screen. Tabs have also long been used for different pages of controls in dialog boxes, as explained in Section 9.4, with examples from 1988 for a research system and 1993 for Microsoft Word (Figures 9.4(b) and 9.5). Tabs for different workbooks of a spreadsheet were introduced at least in Quattro Pro for Windows, release 1.0 in September 1992, followed shortly by Lotus and Microsoft Excel [Seymour 1994]. Tabs have been used for web browser pages since the late 1990s and to help arrange files in code editors since at least Visual Studio .NET in 2002.

The advantage of tabs is that each window is easily accessible, the user can change windows with a single click, and it is easy for the user to scan across all windows. However, this only works when there is a small number of tabs—with many tabs, either the user must scroll to see other tabs, often one-by-one using an arrow button (see Figure 11.5(b)), or else the tabs get so small that the user cannot tell what they are for (Figure 11.5(a)). Another issue is that the user can only see one window at a time, which is a problem if the user needs to compare or cross-reference the contents.

Finally, small devices like PDAs and smartphones only show one application at a time, so there is only one single window shown, and there are interaction techniques to pick which one is displayed, as discussed below.

11.6 Icons

As explained in Section 2.5.1.3, David Canfield Smith (Chapter 24) coined the term *icons* in his 1975 Stanford Ph.D. thesis on Pygmalion [Smith 1977], and Smith later popularized icons as one of the chief designers of the Xerox Star in 1981 (Section 2.4.4). The icon represented a file, folder, or operation managed by the computer. Today, we use the word *icon* more generally for any small graphic that represents something, and may be used on the desktop to represent files and

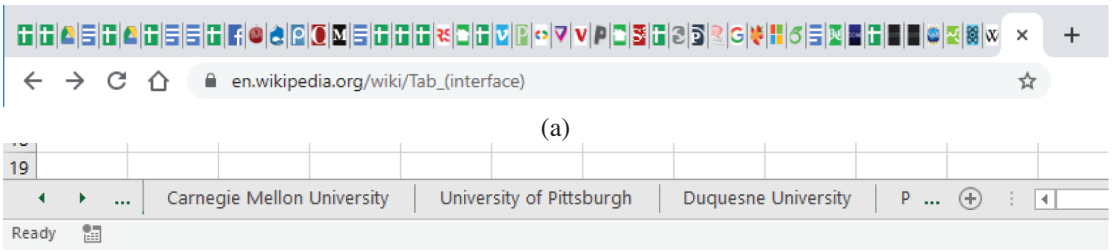


Figure 11.5 (a) Tabs for web pages get tiny when there are too many open in Chrome v. 79.0.3945.130. (b) Excel 2016 provides arrows and ellipses when there are too many tabs. Sources: Screen captures, 2020. Used with permission from Microsoft.

folders, in menus and palettes to represent operations, on physical buttons on consumer electronics, and even as the cursor picture. Icons are used ubiquitously in current window managers for regular computers, tablets, phones, and even watches (see Figure 11.6). Icons in window managers are used to represent an application (like Microsoft Word or Safari, Figure 11.6(a), (b), and (d)), a particular file or folder (like "IxTBook.docx" or "Games," Figure 11.6(c) and (d)), or an instance of a program that is currently running (Figure 11.6(e)).

Icons originally represented objects or operations in an office environment (documents, folders, file cabinets, printers, etc.) (see Figure 11.7). Key interaction techniques for these icons included selecting them, opening the associated object, or moving them (dragging) to another object to perform an operation (like putting them in a folder or printing). On the Star, operations were invoked with a combination of the special keyboard keys (like "move", "copy", "delete", and "open") and the mouse (see Figure 9.1). The Apple Lisa and Macintosh (and most subsequent systems) supported single click to select and double-click to open and drag-and-drop using purely the mouse button, and added dragging to the trashcan as the interaction technique for deleting. One subtlety was that it tried to intelligently decide whether to copy or move—for example, dragging to a folder on the same disk performed a move, but to a different disk performed a copy.

Other early systems used icons as an alternative representation for windows, so that a window would either be shown full size or "shrunk" to an icon. Some of these systems, such as the original Macintosh and Star, left a "shadow" of the icon to show where it came from, whereas other systems (SunWindows and X/uwm) did not. Sapphire was the first system to group the icons into a separate window to serve what is now called a *taskbar* (Section 11.7)—to allow monitoring of the running applications. This design was later also used in Motif (Figure 2.22). The icons in Sapphire could be used to give commands to the windows and to show

Figure 11.6 Many icons: (a) The icon for the Microsoft Word program. (b) Icons on the Apple Watch for various applications. (c) Three icons on Windows for folders and a file. (d) Icons on the iPhone—the bottom row is the iPhone dock or bottom bar, and the row above it contains two applications, then the Games folder, and then an icon for my web page. (e) Windows Taskbar showing running applications. Sources: Screen captures, 2020.

eight pieces of status information about the window and the process running in the window [Myers 1984]. This included percent-done progress indicators (Chapter 6) to show how the job was doing and pictures to show when an error occurred or when the process in the window was waiting for input (see Figure 11.7). Sapphire icons could change to show the application's state, which was also a feature of X where an icon could be a tiny version of the actual window (see Figure 11.7). Most other icons were static, except for small changes like showing new mail (in Xerox PARC's Cedar, where the mailbox icon's flag would go up, and in the Xerox Star, which would show mail in the inbox icon) or in the original Macintosh where the trashcan would bulge when an item was thrown away.

The Palm Pilot (from 1996) and Palm watch (2003) used icons to represent all the applications that could be run (see Figures 11.8 and 3.19), which is also the design used in the iPhone (2007, see Figure 11.6(d)). Other design decisions with icons are whether the user can move them around, whether their shape and size are fixed, and whether the user can give commands to the window using the icons. In the extreme, if icons are just an alternative representation for windows, the user should be able to type to the icon and have the text go to the application running

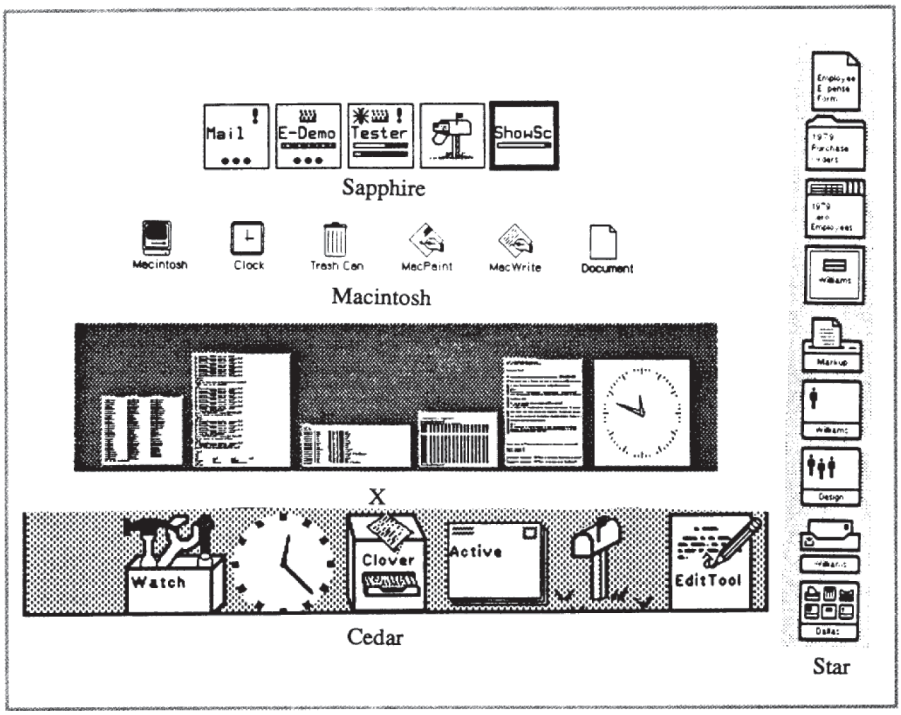


Figure 11.7 Icons from various historical systems: Sapphire [Myers 1984], original Macintosh [Williams 1984], X [Scheifler and Gettys 1986], Cedar, [Teitelman 1985], and Star [Smith et al. 1982b]. Some of the X icons contain the actual text displayed in the window in a tiny (unreadable) font. Source: Fig. 3 of Myers [1988].

in the window. This is allowed in some window managers such as Sapphire, but others limit the commands available when the window is shown as an icon.

11.7 Task Bar and Dock

As mentioned above (Section 11.6), the Sapphire window manager (1983) organized all the icons in a window rather than having them distributed on the desktop or in a folder as in previous systems, and there were icons for all of the running processes. If the icons did not all fit, then the icon window got bigger. Later, one of the window managers for X-windows also provided for a list of the running windows to be displayed in a window.

Windows NT from 1995 and Windows 95 (see Figure 11.9(a)) added the *taskbar* to help users manage multiple open windows (see Chapter 31). Each open window had an icon and text label in the taskbar that the user could click to bring the window to the front or remove the window and only show its icon, as well as other

Figure 11.8 Palm handheld used icons to represent all applications that could be run. The four physical buttons at the bottom and the calculator icon next to the text input area also start applications. Source: Screen capture.

operations in a right-click menu, such as closing the window and its application. Also in the taskbar was the *Start button*, which provides access to all the installed applications, plus other settings such as the control panel and options for turning the computer off. There could also be icons in the taskbar to launch selected applications, and indicators for other status such as the time. Later, the Windows taskbar added other features such as serving as a progress bar for the operations in the window (Section 6.4). The taskbar can be moved from the bottom to be at any edge of the screen through settings, and its size can be adjusted to be one or more rows or columns of entries.

The NeXT computer (see Section 2.4.11) provided a *dock* starting in about 1996 that included icons for frequently used programs (see Figure 11.9(b)), which served the same purpose as the taskbar—it could be used to start applications or bring an application's window to the top. A small ellipsis with each icon showed that the corresponding application was *not* currently running. The dock was brought from NeXT to the Macintosh where it appeared around 1999 in Macintosh OS X (see Figure 11.9(c)). Both the NeXT and Macintosh docks include the icon for deleting files (the trashcan on the Macintosh), which was not in the Windows taskbar.

Figure 11.9 (a) Windows 95 showing icons on the left and the taskbar at the bottom with the "Start" button. (b) NeXT computer's Dock. (c) Macintosh dock as it appeared in OS X 10.8 to 10.9. Sources: (a) Windows 95 (WP:NFCC#4), Fair use, https://en.wikipedia.org/w/index.php?curid=42693963. (b) OPENSTEP, Fair use, https://en.wikipedia.org/w/index.php?curid=19715956. (c) MacOS, Fair use, https://en.wikipedia.org/w/index.php?curid=16466683.

Other features of the current Macintosh dock are that it adjusts the icon size to fit any number of icons, with a zooming feature around the cursor to make small icons easier to see and click on, and that the icons will "hop" up and down when the associated application wants the user's attention.

The iPhone provides a more limited form of the dock as the bottom row of the screens of icons. It includes four spaces for icons, which the user can assign to

their most frequent applications (see Figure 3.26 for original iPhone from 2007 and Figure 11.6(d) for a closeup of iPhone 11 from 2020).

11.8 Decorations for Individual Windows

Another aspect of the interaction techniques for windows is what the decorations on the windows themselves look like. This usually includes special *title lines* at the top and special borders around the entire window. The title line typically shows such global information as the name of the program running in the window, the current directory, the name of a file being edited, and so on. The original covered window manager, Smalltalk, displayed the title lines to look like file tabs (see Figure 2.10), but most other covered or tiled window systems used the full width for the title line. Historically, the title line is usually shown in reverse video (light text on a dark background), as shown in Figures 2.17, 3.21, 5.11, 11.1, 11.2, 11.3, 11.9(a), and so on, except on the original Macintosh (Figure 5.12). The Blit window manager [Pike 1983] was one of the few window managers that did not use title lines. In addition, the title lines and borders may contain other decorations and command buttons and icons, such as a close box or windows menu.

The NeWS window manager from Sun (launched in 1986) [Gosling 1986a, Sun 1986] was one of the first to support windows that could be non-rectangular owing to being implemented on top of a version of the PostScript graphics language [Adobe Systems Inc. 1985, Warnock 2018]. The window manager from SGI in 1989 also supported non-rectangular windows and added partial transparency, leveraging their more advanced graphics hardware.

For single-window systems for small devices, like the Palm and iPhones, there was a trend toward reusing the title bar area for other purposes. For example, on the Palm the title bar looked like a tab and the right side was used for picking the date (Figure 3.18(c)) or showing the battery and connected status (Figure 3.20). Tapping on the Palm title line replaced it with the menu bar (see Figure 7.7(a)), saving space.

However, there is increasingly a move to omit the title bar from modern systems. For example, recent versions of the iPhone apps do not display a title bar, requiring that app designers make it clear with the window *contents* what app is displayed. Similarly, the Google's Chrome browser on Windows (see Figure 11.5(a)) and other browsers, along with other applications like Apple iTunes and Visual Studio Code, no longer display the window manager title bar on regular computers.

Another aspect of the decorations for individual windows is how the *borders of windows* are shown. The older systems like Smalltalk (Figure 2.10), Cedar, and so on just used a thin line, and the Lisa and Macintosh put the Window resize

button in the lower-right corner (Figures 5.11 and 5.12). Sapphire (Figure 11.1) used thicker borders around the window to indicate which window had the input focus (Section 11.3 discusses other ways to display the input focus). Later, Windows 3.0 (1990) used thicker borders around windows so the user could press down with the mouse on any edge to resize the window (Figure 11.2).

11.9 Interaction Techniques for Windows

Section 11.3 discussed various ways that the user can change which window has the *focus*. This section discusses the other interaction techniques that allow the user to manipulate the windows. See also Sections 2.5.1.4 and 11.3 that provide a brief overview of the history of window systems, and my survey from 1988 that covers early systems [Myers 1988], with many of their interaction techniques shown in the "All the Widgets" video [Myers 1990b].

To *create* new windows, on computers the user typically must double-click on an icon or filename for the data file or application, or single click if the icon or name is shown in a menu, taskbar, or dock. On PDAs and phones, the user can tap on an icon or use one of the hard buttons on the Palm PDA (Figure 11.8). Microsoft Windows systems since Windows 95 have provided a list of all applications in the Start menu in the lower left. Macintosh has allowed users to type in the name of the file or application using the Spotlight feature since Mac OS X 10.4 released in April 2005. Windows has always allowed users to type the name of a program into the "Run" dialog box, or search (slowly) using Windows search, but with Windows 10, released in 2015, Cortana was introduced that provides more general searching and natural language processing for the operating system. From inside an application, often an "Open" or "New" menu item will launch a new window for the same application on a different document. For tiled window managers like Star, Cedar, and Andrew, the system would automatically pick a place for the new window. To *split* a window into two panes, the original Lisa (see Section 23.9), Microsoft Word up until before version 365, and current versions of Acrobat display a small rectangle at the top of the vertical scrollbar, which divides the current window into two panes.

To *delete* or *close* a window, the user would typically click on an "X" or red dot icon displayed in the title line, or else select "Close", "Quit", or "Exit" from a menu for the window or from a menu for the application itself. As with all other menus (Section 7.3.11.4), there are often keyboard accelerators for these commands, like Control-q for Quit or Alt-F4 on Windows to close. The Xerox Star had a "Close" menu item on each window, and the special "Delete" key on the keyboard was used to delete the file for the window. The Apple Lisa introduced dragging an icon

to the trashcan to delete the corresponding file (and the window for that file, if open). To delete a split window (and return to a single window), the user can usually double-click on the separator or use a command.

For covered window managers, an important operation is to bring the window *to the front* so that it is not covered by any other windows. Typically, this is the same operation that gives the window the input focus (e.g., clicking on it), but some window systems have an explicit command for bringing a window to the front. Older window managers had a command to send the window *to the back*, but modern window systems omit this operation and instead the user iconifies the window.

To *move* a window, almost all systems allow the user to drag the title line. This may be more limited in systems with tiled windows (Section 11.5), where normally windows can only be moved to another column. Some window systems, such as Sapphire and Microsoft Windows, have a Move command in a menu, which allows the window to be moved by the mouse or keyboard arrow keys. The feedback during the move was originally just an outline box of the window's size, but eventually switched to moving the actual window itself. Some systems allow the window move to be aborted while in progress, typically by hitting the Esc keyboard key, but no system has an "Undo" for general window moves. Another issue is whether a window can move to be partially or entirely off-screen (most do allow this).

There are various ways to *resize* a window. The original window managers, like Smalltalk and Interlisp, had a resize command in a menu that allowed the user to drag out a new rectangular region for the window (called Reframe in Smalltalk), and a different command allowed a single corner or size to be adjusted. The Apple Lisa and Macintosh introduced an icon at the lower right corner of windows that provided for window resizing, but only from that corner. Microsoft Windows since at least OS/2 Presentation Manager in 1988 has provided thicker borders around the windows, which the user can press on to resize from any side or corner. Some tiled window managers (and most split windows) allow the seam between the columns to be adjusted, changing all the windows sizes, and sometimes the border between two windows could be adjusted to change the size of just those two windows (this was provided by Bravo in 1974, Section 2.4.3.2). The size of some windows, such as most dialog box windows and alerts, cannot be changed on most systems. In addition, windows on mobile devices are generally the full size of the screen, but they might adjust their own size if the keyboard is popped up or the device is rotated 90 degrees.

Recognizing that often the size of a window would be too small, most window managers, dating back to at least Interlisp in 1980, have an operation to make a window *full screen* or full size, often called *maximize*. The old size and position are remembered, so a single operation can be used to make it go back to its original

size. Window managers since at least the Macintosh provide this as a button on the title bar. Double-clicking on the title bar often also toggles the window to be full screen or back to regular size. This operation was even supported by the Cedar tiled window manager, which would cause all the other windows in the column to be removed and only show their icons.

For window managers that have icons or a taskbar, there are often operations to *shrink* the window down to the icon (also called *iconify* or *minimize*) and to expand from the icon to normal size (*restore*). In the Xerox Star, the "Close" command shrunk the window to the icon. The Macintosh has always provided a button that would reduce a window to its icon, but the look of this button has changed over the years. In Windows 2.0 and 3.0 (Figure 11.2), the title bar had a down-arrow for iconify and up-arrow for full screen (and the hyphen on the left would bring up the Windows menu), and Windows 10 now uses a hyphen for iconify, a square for maximize, and an "X" to close. The Andrew system did not have icons per se, but windows could be shrunk down to just show their title line.

The Macintosh and Windows have commands to minimize *all* open windows, and *show the desktop*, since often the user needs to find an icon on the desktop that is hidden by the windows. This has been an explicit menu command on the Macintosh and on Windows since Windows 95 and is also triggered by clicking on a tall skinny button at the right edge of the taskbar. On the Palm, iPhone, and Android, there is a home button to close the open app and return to the home screen, showing all the icons.

Microsoft Windows 7 in 2009 introduced what was originally called *Aero Snap* (and in Windows 10 is just called *Snap*) as shortcuts for maximizing, resizing, and moving windows [Thurrott and Rivera 2009]. If you drag a window and hit it on the top of the screen, then it will be maximized. If you instead grow from the top or bottom edges of the window and hit the top or bottom screen edge while dragging the size, then the window will resize to fit the screen vertically, but will stay the same width. Finally, moving the window and hitting the cursor on the right or left edge of the screen will make that window be full height, with a width of one-half of the screen. Another feature is *Aero Shake*, where grabbing the title bar and shaking the window left and right minimizes all *other* windows, or restores them.

Other notable window manager commands include *drag-and-drop* of icons from one folder to another, and of content, like text or pictures, from one application to another (discussed in Chapter 10). Drag-and-drop typically requires that both windows be open and not completely occluded, but Windows provides that moving the pointer to an entry in the taskbar (while continuing to hold the mouse button down) causes the corresponding window to come to the front, so the user can complete the drop operation in it.

Finally, as there get to be more and more files and applications, it becomes difficult to remember where the corresponding icon or file is located, so window managers have added a search feature whereby the user can type part of the name (or even content) of the file or application. Windows has long had a *search* for files, and introduced searching inside files in Windows 95, but these are slow. The Macintosh introduced *Spotlight Search* for this with Mac OS X 10.4 in April 2005, where it is invoked from a looking glass icon in the menu bar and uses a pre-built index so searches are much faster. A similar feature appeared on the iPhone in v3.0 in 2009 by swiping to the right from the first home screen, and later by swiping down from the center-top of any home screen.

Of course, most of these operations have keyboard accelerators. For example, on Windows a keyboard key has the Windows Logo on it, and this key plus another key performs a window operation—for example, Windows-right-arrow snaps the current window to the right. On Macintosh, Command-F3 shows the desktop. Modern systems have a wide variety of commands to find and organize windows.

11.10 Collections of Windows

As early as 1986, researchers were already identifying that users were spending significant time re-arranging and minimizing windows in order to see the desired windows. One proposed approach was to use tiled windows [Bly and Rosenberg 1986], but this breaks down where there are more than a small number of windows.

Austin Henderson and Stuart Card at Xerox PARC described the *Rooms* system in 1986 as an alternative approach [Henderson Jr. and Card 1986, Card and Henderson 1987], and it became enormously influential. They studied various models for the data describing the number of windows used together for various tasks (the users' *working set* of windows for those tasks), showing that between two and ten windows were used together. Then, they built a system with a collection of *rooms* or *virtual workspaces*, each with a unique background and a collection of windows that would be used together for a particular task (see Figure 11.10). The windows for an application were allowed to be in multiple virtual workspaces with different sizes and positions—for example, the email application might be useful for both programming and management tasks. Clicking on a *door* transited to a different room. There was also an overview showing all the rooms and their windows.

This concept, which is also called *virtual desktops*, was quickly adopted by other research and commercial window managers, first for various window managers for the X Window system starting around 1989, and later for the Macintosh, where it is called Spaces, and was introduced around 2006. On Windows it is called Multiple Desktops, but it was not supported natively until Windows 10 in 2015. However,

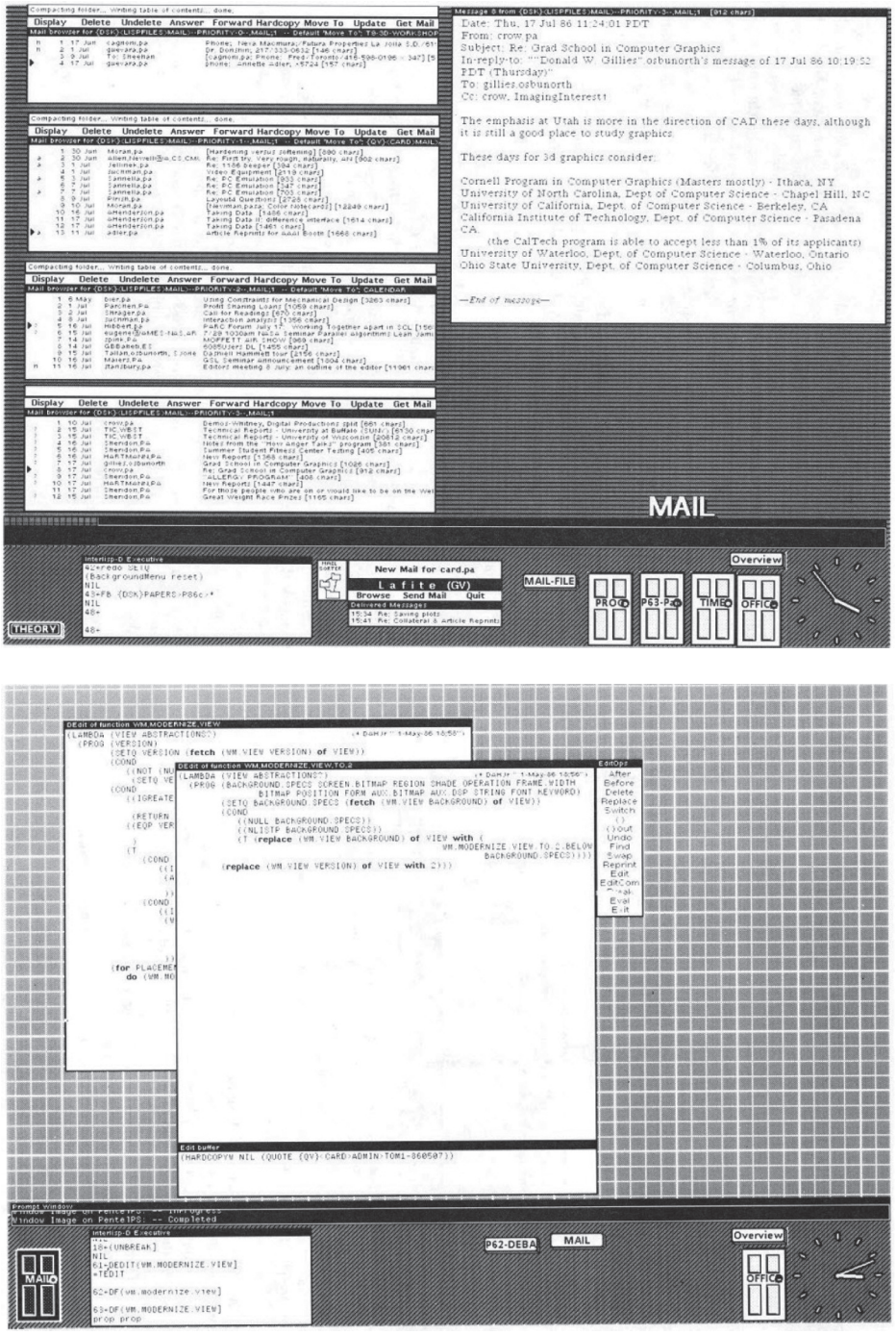


Figure 11.10 Two "Rooms"—top for email and bottom for programming. Source: Fig. 9 of Henderson Jr. and Card [1986].

these implementations usually do not allow the same instance of an application to be open in different desktops (each window can only be on one desktop at a time).

11.11 Alternatives

Although virtually all full-size and mobile systems use the conventional desktop with windows and icons, as described above, there have been research systems that investigated alternative layouts with different interaction techniques for finding and switching among different tasks and windows. The most prominent are systems using zooming and panning and ones based on a timeline metaphor, along with a few others.

11.11.1 Zoomable User Interface

In a *zoomable user interface* (ZUI), also called a *multiscale interface* or *zooming interface*, the screen is represented as an almost infinite sheet on which the user's tasks or windows can be placed. Since there is plenty of room, the contents usually do not overlap. When the user wants to find something, they zoom out to get an overview and then zoom into the information that they want. The metaphor is that the user's content is on a stretchy sheet. Like the desktop metaphor, the thought is to take advantage of the user's spatial memory and natural geographical navigation to make finding things easier. Sometimes, there are multiple views, so the user can see an overview of the space and a zoomed-in (detailed) view at the same time. A contemporary use of a ZUI is Google Maps, where the user can zoom out to see the whole world or zoom in far enough to see just a few buildings. Another current example is the Figma prototyping tool, where users can draw pictures of screens on different parts of large sheets (see Section 20.9.2). However, the original ZUIs supported arbitrary information on the zoomable sheet, not just map or picture data. The primary interaction techniques for ZUIs are *zooming* in and out (where "in" makes the contents bigger) and *panning* (moving sideways at the same resolution).

Often, these systems support what is called *semantic zooming*, where different versions of the content are shown at different resolutions. For instance, if a calendar is on the screen, when zoomed out so it is small, maybe only the current date is shown, when zoomed in a little, the current week might be shown, and zooming further in shows details of the current day. This is called "semantic" because the contents switch to different representations entirely, not just showing more detail on the original representation. (Google Maps does this as well—adding and removing features entirely at different zoom levels and switching to a 3D view of buildings when zoomed in close enough.)

Except for maps and homogeneous images, ZUIs have not caught on due to the difficulty of finding things. With many items, zooming out to see them all makes

many things invisible, and zooming in results in a loss of context. Other barriers are the difficulty of manually placing new items and the difficulty with rearranging or sorting items.

11.11.1.1 SketchPad

Like so many other interaction techniques, ZUIs go all the way back to SketchPad from 1963 (Section 2.4.1). The graphics was displayed on scalable "sheet" that had a "range of magnification of 2000 [which] makes it possible to work, in effect, on a 7-inch square portion of a drawing about ¼ mile on a side" [Sutherland 1963]. The interaction technique to move around (pan) and zoom used four physical knobs located below the display (see Figure 2.8).

11.11.1.2 Spatial Data Management System

As discussed in Section 2.4.5, the SDMS [Bolt 1979, Herot et al. 1980] from the Architecture Machine Group at MIT (now the MIT Media Lab), with a commercial version from Computer Corporation of America, was the first system to have a general-purpose ZUI. As shown in Figure 2.13, it provided an overview on a small nearby screen (called the *world view*) and a full-size view on a large projection screen in front, with another screen for a detailed view and interactions. The user navigated using physical joysticks embedded in the arms of the chair, and the monitors at the sides were touchscreens. SDMS was one of the first systems to support semantic zooming, although they did not use that term.

11.11.1.3 PAD, PAD++, and Jazz

In the 1990s, some researchers revisited the idea of ZUI with the Pad [Perlin and Fox 1993], Pad++ [Bederson and Hollan 1994, Bederson et al. 1996], and then Jazz [Bederson et al. 2000] systems, which took advantage of better hardware and displays and also new graphics algorithms for faster and smoother zooming and panning. The example applications included a drawing editor, a web browser (with links dynamically loading new pages smaller and to the right of the current page—to visualize the source-destination structure), file directory viewer, and so on. The interaction techniques for Pad++ [Bederson and Hollan 1994] included using the left mouse button for application-specific actions on the object under the mouse, such as selecting or drawing in a graphics editor and using the middle button to zoom in and the right button to zoom out. Since Pad++ zoomed in and out around the current cursor position, the user could control the zooming dynamically by moving the mouse while zooming in and out. This replaced panning with moving sideways while zooming out and then zooming in. In Jazz, this was replaced with

letting the user "pan with the left mouse button, and zoom with the right mouse button by dragging right or left to zoom in or out, respectively" [Bederson et al. 2000].

11.11.2 Timeline Systems

A completely different alternative arranges content by *time* instead of spatially, taking advantage of the user's memory of approximately *when* events happen, in what is sometimes called a *timeline view*. Another key principle of timeline systems is that *all* content is mixed together on the timeline, including files, pictures, emails, web pages visited, and so on, rather than having each kind of information siloed into its own application. Thus, if I created a document, received an email, took a picture, and then visited a web page, representations of these would be contiguous in the timeline view. Contrast this with a regular system, where sorting by time is only available within each application, and with a conventional hierarchical file or mail system, where it is only available within each folder.

One of the first timeline systems was *Lifestreams* [Fertig et al. 1996] by David Gelernter's group at Yale.[1] As shown in Figure 11.11, it allowed the user to quickly scan through content by moving the mouse back and forth along the stream, or to scroll back through history using the scrollbar at the bottom. One of the interesting interaction techniques in Lifestreams were that the "Find" command was actually a query that created a "substream" of all matching documents; another technique was a "transfer" command to share documents with others. This idea was further explored in a book [Gelernter 1993] and then by a commercial company named Scopeware.

Another important example of a timeline view is the TimeScape system by Jun Rekimoto [1999] at the Sony Research Lab. Timescape was designed to replace the conventional Macintosh or PC desktop by arranging all items by time (Figure 11.12). This was called *time-machine computing* since the user could move into the past or even the future (e.g., to show future calendar events or to put a reminder at a future time). This idea was commercialized by Sony in the SONY VAIO XR notebook PC with its TimeView user interface.

A somewhat less radical approach was the *Stuff I've Seen* system [Ringel et al. 2003], which provided an interactive timeline-based visualization of search results. To make it easier to find items from the past, the system also showed public and personal *landmark events* such as news items and appointments. This view augmented the conventional desktop rather than replacing it.

1. David Gelernter is also famous for being an artist and as a target of the Unabomber where he was injured and lost a hand.

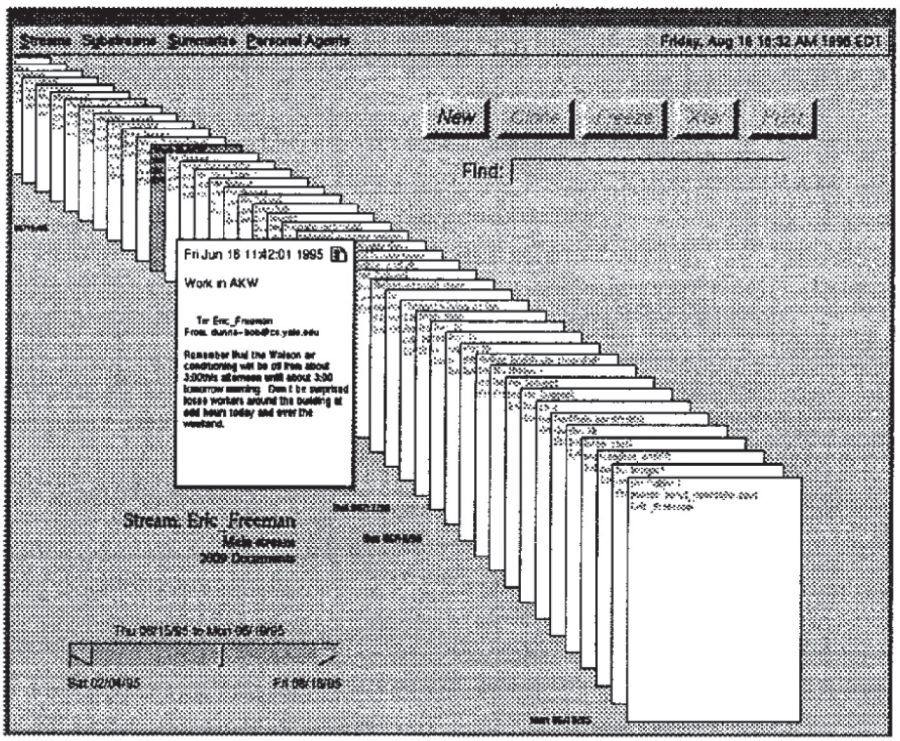
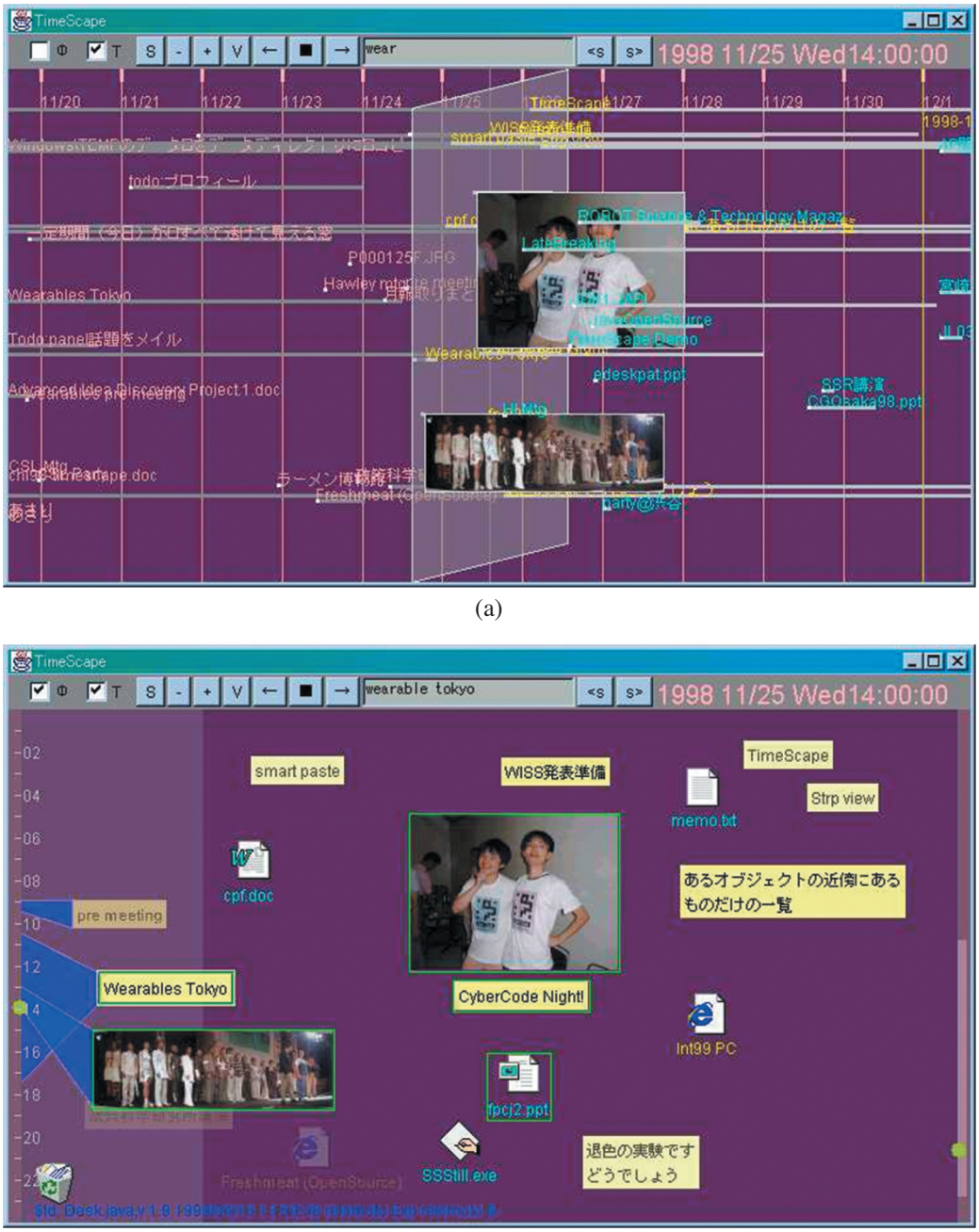


Figure 11.11 Lifestreams. Source: Fig. 1 of Fertig et al. [1996].

11.11.3 The Open Data Model

A completely different model for accessing information has been proposed, where there are various standard formats for data and a variety of programs know how to perform various operations to the *same* data. I called this the "Open Data Model" in 1998 [Myers 1998c], but others have used different names like Michael Dertouzos's *e-forms* [Dertouzos 1997]. Some formats, like PNG for images, come close to allowing any program to operate on it, but still some editors like Adobe Photoshop use entirely different and proprietary formats for their images. Data types for textual documents, email messages, calendar events, contact lists, and so on all have interchange formats, but the Open Data Model requires that there not only be a shared format on the disk but also a shared format in memory so that multiple programs can edit the data, possibly even at the same time. In this way, the user could select, and more importantly compose together, editors for different data and use the least complex editor that was appropriate for each task. A research system that strived for this goal was the Andrew system (Section 2.4.12), and Microsoft Window's COM and OLE systems were designed to support applications built out of components



(a)

(b)

Figure 11.12 TimeScape showing (a) the timeline view and (b) zoomed in for finer details. Source: Figs. 4 and 2 of Rekimoto [1999].

that would interoperate on the same data file. However, vendors still prefer to use their proprietary formats and lock users into using their tools, and users prefer to mainly learn and use only one tool for each kind of data.

The web comes closer to this model than conventional desktops, with the standard data types for HTML, XML, JSON, and the DOM, and Tim Berners-Lee argued for this in what he called the *semantic web* in 1999, where all web-based data would be accessible to be analyzed [Berners-Lee et al. 2001]. Although web services provide significant data, each source has its own API, and web applications like Google's Gmail still keep their data separate and in a proprietary format.

11.11.4 Microsoft Bob

A short-lived alternative to the desktop, and one of the first attempts at an "intelligent assistant," was *Microsoft Bob* [Feldman 2016]. It replaced the desktop and program manager with a representation of a 3D house, with different activities taking place in different rooms, and a variety of cartoon characters trying to be helpful (see Figure 11.13). Bob was released in March 1995 and discontinued in early 1996. Users (including the novices it was aimed to help) generally found the assistants irritating and not particularly helpful, and finding what was wanted turned out to be no easier than the original desktop. The key legacy of Bob was Microsoft's Clippy intelligent agent (see Section 18.5), which was also not very successful.

Figure 11.13 Microsoft Bob. Source: By Microsoft, Fair use, https://en.wikipedia.org/w/index.php?curid=45416346.

11.11.5 Other Organizations

There have been a number of other approaches to finding and operating on a user's information and files. The Apple Macintosh and iOS Spotlight search textually searches through all files, calendar events, web pages, email messages, images, and so on for the entered words and thus provides an alternative to using the desktop and windows to navigate and find items. This was originally available (in 2005) only for searching files and folders; other data types were added later. Note, however, that while these search systems replace the desktop with a different way to *find* the information, they generally still revert to a conventional window manager when the user wants to view or edit the data, such as launching a conventional calendar program or contact manager.

Polo Chau and I investigated *associative search* for information in the Feldspar system [Chau et al. 2008]. Rather than having to remember the name, textual content, or date of an item, the user could instead remember what an item is *associated with*, such as "find the file from the person who I met at an event in May." Feldspar supports finding people, files, web pages, calendar events, and so on (see Figure 11.14).

It could also be argued that the hypertext/web model is also an alternative to a desktop since users get to files, web applications, and other content—and switch tasks—by clicking on links rather than using icons and windows. The interaction techniques for hypertext are covered in Section 2.5.1.8.

Finally, the emerging speech interfaces on all platforms can bypass the conventional direct manipulation approach to accessing individual information items—and even replace a window interface for operating on the data (see Chapter 18). For example, a user might say "When is my meeting with the CEO?" or "Add a meeting with my group at 3:00pm," which a speech system might handle entirely by voice, with the user not using the desktop or any windows.

11.12 Summary of Considerations for Desktops, Windows, and Icons

The previous sections discuss a variety of design choices for windowing systems. Here is a summary:

- Is the functionality organized as applications with their own data (as in all conventional desktop and smartphone systems) or in a more data-focused [Myers 1998c] way?
- The organization of windows: whether they can be overlapping, tiled (into *panes*), tabs, or just one at a time (single window).

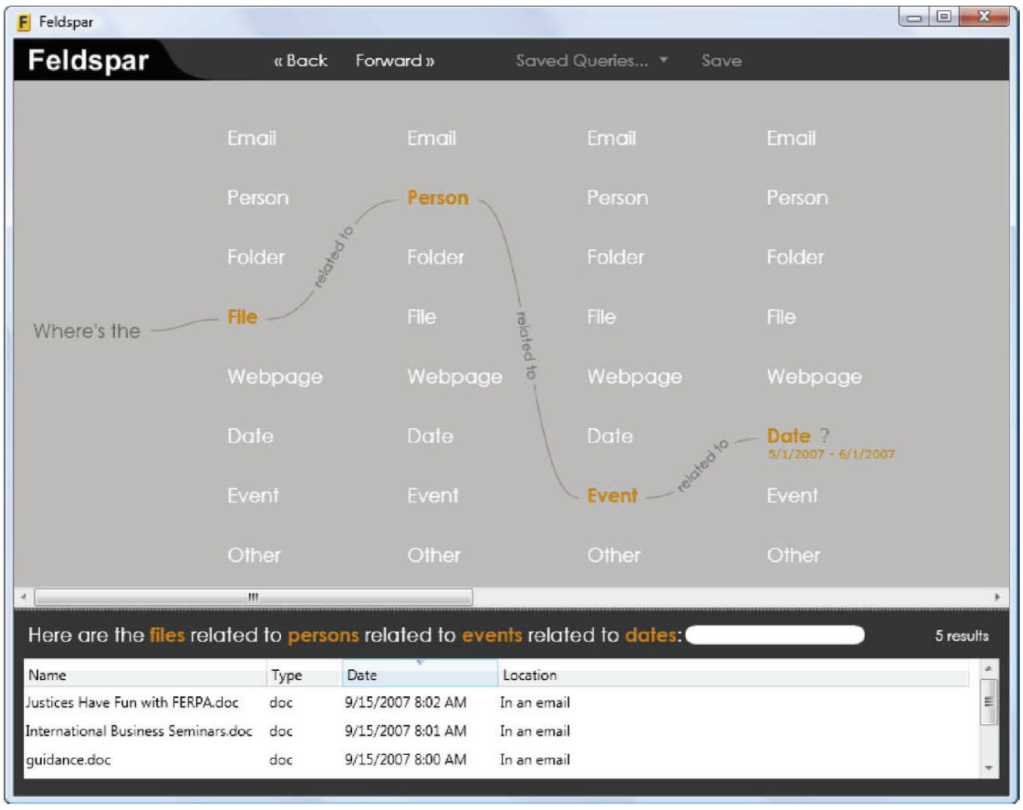


Figure 11.14 Feldspar allows finding items by association. Source: Fig. 1 of Chau et al. [2008].

  - If there are overlapping or tiled windows, whether they are arranged spatially (like most window managers), temporally in a timeline, or on large zoomable sheets.
- What decorations are on the windows, such as title bars and borders, and what operations are available using those decorations. Also, can windows be non-rectangular shapes or transparent, and does this affect the decorations?
- What appears behind the windows, on the "desktop?"
- Whether icons are supported for the applications and/or windows.
  - If so, how are the icons arranged?
  - Is there a taskbar/dock or other place where icons are collected?
- How the user specifies the input focus for the various input devices. In particular, where does keyboard typing go, and which window's content is scrolled (if these can be different, as in most current desktop systems)?

- The specific interaction techniques available for manipulating windows and icons.
- Other operations that help organize windows, especially when there are many, such as *rooms*, *spaces*, or *virtual* or *multiple desktops*. If so, can the same window exist in multiple rooms?
- Are there global operations across all windows and applications, like Spotlight search or a speech interface, and if so, how does it get at application data and how does the user view and manipulate the data if needed?

11.13 Chapter Conclusion

The basic design for windowing systems emerged in the 1980s and has changed surprisingly little in the subsequent 40+ years, even though the number of objects and applications that users manipulate has grown by orders of magnitude. Smartphones and other small devices have adopted many of the desktop ideas, such as icons launching applications, but have screens that are too small for multiple windows. Web applications, on the other hand, have avoided icons and overlapping windows, relying more on hyperlinks, typing of URLs, and tabbed windows. Future interfaces, such as speech (Chapter 18) and virtual reality (Chapter 12), have not yet established a common way to interact with separate tasks, but hopefully they will develop interaction techniques that go beyond what is available today, which arguably do not scale to the millions of items that users need to deal with.

CHAPTER 12

Interactions in 3D, Virtual and Augmented Reality, and Games

12.1 Introduction

The interaction techniques discussed in the previous chapters have primarily been for manipulating in one dimension (like scrollbars and sliders) or two dimensions (2D) (like selecting with a mouse and text and graphics editing). Interaction techniques for three dimensions (3D) are not nearly so well developed. In most cases, the techniques for manipulating 3D objects are displayed on a normal flat (2D) screen, but a few interaction techniques discussed in this chapter are for manipulations in the real world (e.g., for robotics) or a virtual 3D world. This chapter discusses why interactions in 3D are much more difficult, and some approaches that have been tried over the years. However, there are fewer situations in which users *need* to interact in 3D, and little standardization of interaction techniques, so much future research and development is needed. As with other topics in this book, a single chapter cannot cover 3D UIs in depth, but there are entire textbooks on this topic (e.g., Barrilleaux [2000] and LaViola et al. [2017]).

How to produce a 3D scene on a computer (called *rendering*) is generally the area of *computer graphics* and is not covered in this book (see a graphics textbook, e.g., Hughes et al. [2013]). Sometimes, computer graphics is used to create special effects for a movie or television show, in which case the result is not interactive and has no interaction techniques. This chapter focuses on *interactive* 3D computer graphics UIs.

12.2 Definitions

A *three-dimensional interaction technique* (3D-IxT) is any way to manipulate a computerized 3D object, or alternatively, to use three or more dimensional motions to do any kind of computerized interaction. The first part of the definition includes

using a one-dimensional (1D) slider or a 2D mouse to manipulate an on-screen 3D shape, and the second part includes waving one's hand in the air to move a 3D character on a screen, or even to point with a finger in the air at an on-screen (2D) keyboard.

3D graphics are often rendered on a conventional computer monitor or television screen (e.g., for games). Sometimes, the graphics might be projected onto a large display such as an entire wall or even multiple walls of a room, which then is often called a *cave* virtual environment [Cruz-Neira et al. 1992]. Other times, the content might be displayed on a special *head-mounted display* (HMD) that can either block the user's view of the real world (Figure 12.1) or else can cause the image to be super-imposed over a view of the real world, as with Google Glass (Figure 12.2) or the Microsoft HoloLens (see Chapter 38). Often HMDs, especially when the user

Figure 12.1 Researchers with the European Space Agency in Darmstadt, Germany, equipped with a VR head-mounted display and motion controllers. Source: By ESA, CC BY-SA 3.0-igo, https://commons.wikimedia.org/w/index.php?curid=62840930.

Figure 12.2 Google Glass superimposes an image that users see with their right eye. The side of the device is a touchpad. Source: By Loïc Le Meur—Flickr: Loïc Le Meur on Google Glass, CC BY 2.0, https://commons.wikimedia.org/w/index.php?curid=26050963.

cannot see through them, will have a *head tracker* so that the image shown can change as the users move their heads. Finally, the display may actually be a *3D image*. Examples include a regular screen where the user wears special glasses or goggles that makes the image look 3D (usually by showing a slightly different image to each eye, which is called *stereoscopy*), volumetric displays where the image is actually projected onto different points inside a sphere or cube, true holographic displays that provide the image in the air in 3D, and so on. Examples of current HMDs are the Google Glass (Figure 12.2), the Quest from Facebook/Meta (formerly from Oculus), and the Microsoft HoloLens. There are also ways to make an HMD by mounting a smartphone in front of one's face using a case, which might even be made out of cardboard (Figure 12.3).

If the user only sees an artificial or simulated world, such as when in a cave or non-see-through HMD (Figure 12.1), this is called *virtual reality* (VR). If the user can see the real world *and* a computer image, either because the HMD is see-through (like Figure 12.2) or because a non-see-through HMD has a camera that shows the real world in front of the user, then this is called *augmented reality* (AR). Another example of AR is the *Pokémon Go* game, where a smartphone camera shows the real world in front of the device but a computer-generated character is added to the

Figure 12.3 Google Cardboard holds a smartphone (here visible at the right in yellow) in front of one's eyes. A hole in the back allows the phone's camera to work for AR uses. Source: By Evan-Amos, Public Domain, https://commons.wikimedia.org/w/index.php?curid=45580283.

scene (Figure 12.4). In *mixed reality* (MR), the user can interact using real objects in a virtual world, like picking up a real water pitcher to give a virtual character a drink. Sometimes the term *extended reality* (often abbreviated as XR) is used to cover all of these concepts. However, these terms are in flux and their meanings seem to be evolving.

Some 3D interaction techniques take advantage of the user's *pose*, which means the particular way of holding body parts. For example, a pointing operation may be triggered by a sensor detecting that the user's index finger is in a pointing pose. Sometimes a whole-body pose may be relevant, and not just the hand. In contrast, a *gesture* is a movement through space and time where the path of movement is relevant, which may be a sequence of poses. For example, the user might make a "Z" gesture in the air with their finger. More details on pose and gesture recognition and their interaction techniques are given in Chapter 14.

12.3 Why Are 3D-IxTs Difficult

In 2D, graphical objects can be moved around in X and Y and rotated (see Chapter 10). One might think that adding one more dimension, to go to 3D, would only increase the complexity by one, whereas in fact there is typically at least four times the complexity! This is usually expressed as *degrees of freedom (DOF)*, which are the number of independent parameters that control the object's configuration. With 3D, to position an object requires specifying *six* parameters, so it has

Figure 12.4 *Pokémon Go* augmented reality (AR) game. The green character and red "Poké Ball" are superimposed over the camera view of the machinery and ground. Source: Pokemon Go (WP:NFCC#4), Fair use, https://en.wikipedia.org/w/index.php?curid=56369923.

6DOF. In addition to the X, Y, and Z position in space, there are also three different ways it can be rotated, one along each axis, which are called *roll*, *pitch*, and *yaw* (see Figure 12.5). It is difficult to use the 2D parameters that come out of all conventional pointing devices, like a mouse or touchpad, to control the required 6DOF.

To make matters worse, whenever working in 3D, one needs to also specify from where the object will be viewed. In 2D, the user needs to control 2D scrolling to put the object of interest in view (Chapter 5), but just like for positioning the object itself, in order to put an object in view in 3D the user must specify six parameters. This is called the *camera angle* by analogy with setting up a still camera or movie camera to take a picture of a real scene. The virtual camera (and hence the user) can view the object from any side and can be close or far away from the object. There is significant research on having the system automatically choose a camera angle, but usually there must be interaction techniques for setting the camera view in the 3D space. Thus, the user must specify 12 parameters to set up both the object in space and the camera so that the object can be appropriately seen. Sometimes,

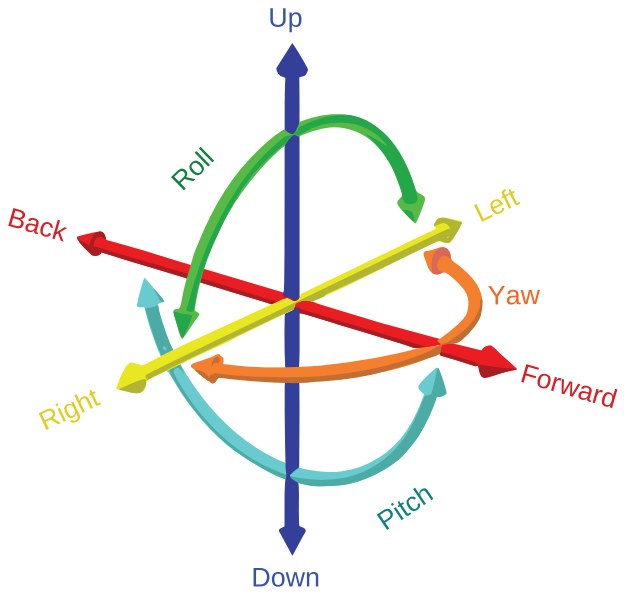


Figure 12.5 The six degrees of freedom (6DOF) for 3D objects. Source: By GregorDS, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=38429678.

conventional scrollbars and zooming techniques are used, but often, custom 3D interaction techniques are involved.

Another issue is that the objects being designed and positioned in 3D are often significantly more complex than 2D objects, with complex coloring and textures, lighting and reflectance, hierarchical shapes (like a character with a head, torso, and limbs that all may need to be positioned in space), and so on. While conventional menus and other widgets are often used to set these (as for 2D graphics, Chapter 10), in other cases custom interaction techniques have been designed, as discussed below.

Related to the previous two points, it can be very difficult in 3D to *select* objects. First, one may need to adjust the view so the object can be seen. Second, the situations in which one object obstructs the view of another is much more complex than in 2D, when it is just due to being covered. Third, if it is a complex, hierarchical object, it may be difficult to identify the correct part. Finally, as will be discussed below, pointing devices in 3D are generally much less accurate than those for 2D.

A further difficulty is that widgets that work in 3D are difficult to design. Many 3D scenes are designed to be photorealistic, and the graphics for the widgets can interfere with graphics in the scene. The widgets themselves may be in the way, potentially occluding the content of interest. And for widgets like selection

handles, it is not clear how to make sure they are visible and distinguished from the graphics in the scene—in 2D, selection handles are usually drawn on top of all graphics as distinguishable squares (Section 10.5.4), but this strategy does not work in 3D.

Additionally, although rendering speeds continually improve, it is possible that a 3D system may not be able to provide interactive speeds for edits, so interaction techniques may need to account for a lag in the refresh.

Finally, even though it would seem that 3D is "natural" for people to think about since it corresponds to the real world, in fact, 3D reasoning (also called *spatial reasoning*, *spatial visualization ability*, or *visual-spatial ability*) is quite difficult and is often used in IQ tests (see Figure 12.6). Often, interacting in 3D requires the user to perform reasoning tasks to predict what a shape will look like from different viewpoints or to select what kind of transformation will change the current shape into the desired one.

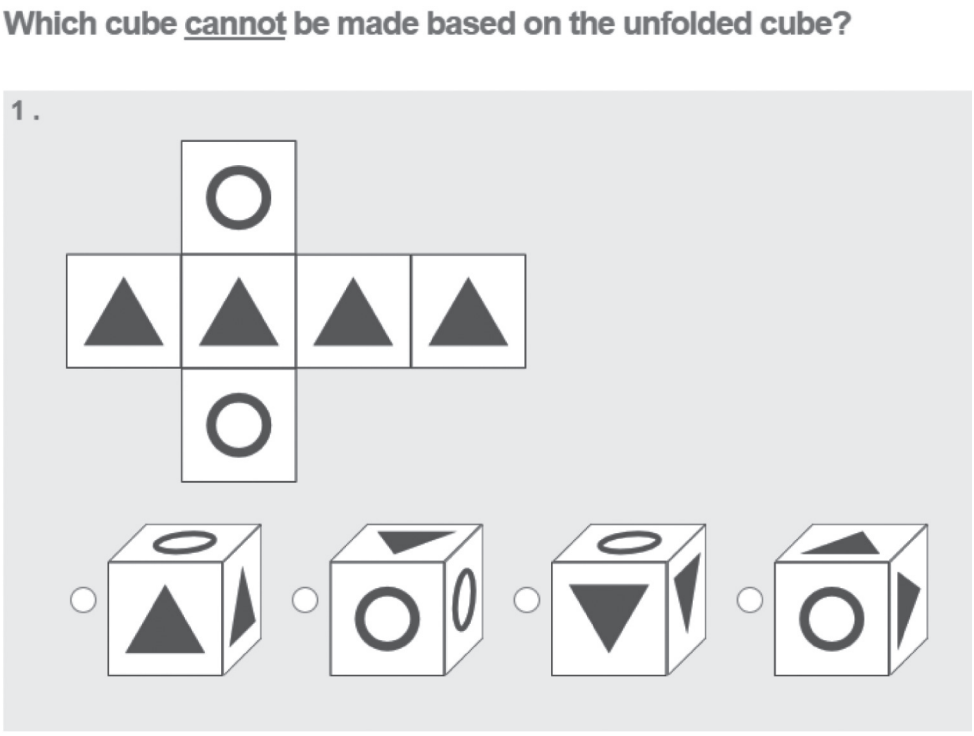


Figure 12.6 Example spatial reasoning test questions. Source: Image from [123test n.d.] spatial reasoning test.

12.4 Input Devices for 3D IxTs

Often, input for 3D uses conventional 1D and 2D input devices, such as physical knobs and a mouse or touchpad. In that case, the interaction techniques must map the 1D or 2D input into the required 6 degrees, see Section 12.5.2 below.

In other cases, a 3D or 6D pointing device is used. These have the same considerations as discussed for 2D devices (Section 4.14): speed of selection vs. accuracy, as well as fatigue (since many of these require the user to hold their hand in the air).

12.4.1 1D and 2D Input Devices for 3D

An extension of the original Sketchpad system (Section 2.4.1) from 1963 supported 3D drawing and manipulations using the 2D lightpen and the collection of knobs and switches (see Figure 2.8) for the various rotations [Johnson 1963]. Although conventional computers have a variety of techniques for 2D input primarily for positioning (Chapter 4), it is rare to have any kind of knob (Section 5.3) or joysticks (Section 4.7). On the other hand, game controllers (including those that can be connected to computers) often provide a variety of joysticks and sometimes knobs, which can be used for rotations and other translations (see Figure 4.8).

Controllers for use with VR and AR devices (see Figure 12.1) also often come with handheld controllers containing a variety of joysticks, buttons, and sometimes knobs, which are designed to be usable without looking at them (since, at least with non-see-through HMDs, the user cannot see their hands).

12.4.2 Early 3D Position Sensors and HMDs

The hardware that could track all three coordinates in the air dates back at least to the 1960s. Sometimes, these devices are call a *bat* [Ware and Jessome 1988] since it is like a mouse that flies (in fact, the German word for bat is *Fledermaus*, literally "flying mouse"). Possibly the first interactive device for 3D input was the Lincoln Wand from around 1966 [Buxton et al. 2005]. The first head tracker on an HMD for computer graphics is attributed to Ivan Sutherland who in 1968 tried both mechanical and ultrasonic sensors [Sutherland 1968]. The device was quite heavy, so it was attached to the ceiling, and was called the *Sword of Damocles* since it loomed over the user's head (Figure 12.7) [Izadi 2016].

One of the first popular 6D input devices was the *Polhemus* company's product, which was originally shipped in 1969 to track the position of a military pilot's head and used magnetic fields emitted by one box and sensed by another [Polhemus 2020]. For computer sensing, one box would be worn on the user's arm or back of the hand, which was then tracked in the air when it was within a few feet of the other

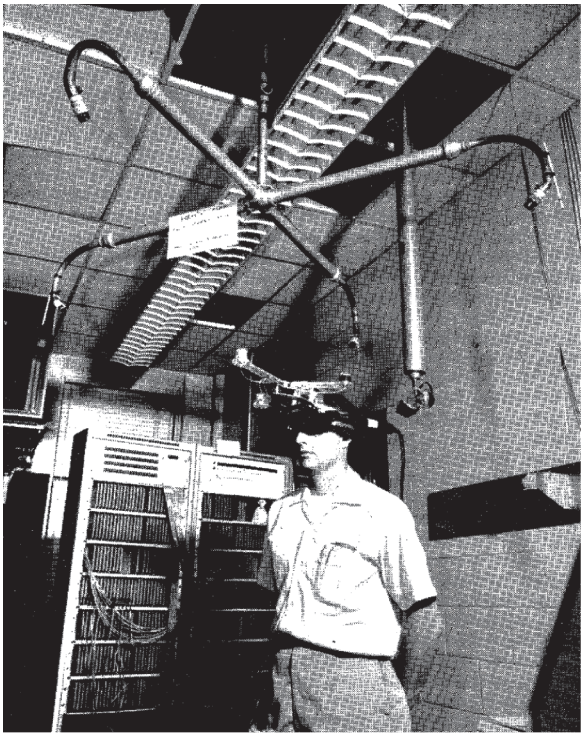


Figure 12.7 Mechanical apparatus for tracking the user's head from 1968, colloquially called the *Sword of Damocles*. Source: Fig. 4 of Sutherland [1968].

box, which was stationary. One famous early use of this tracker was for the *Put-That-There* system [Bolt 1980] from the MIT Architecture Machine Group (using some of the same hardware as the spatial database management system, Section 11.11.1.2) for pointing in the air to the wall display (see Figure 18.4). Other uses took more advantage of its 3D nature. It was also often mounted on a glove that could measure the user's finger poses (see Section 12.4.3), and sometimes multiple sensors would be used at the same time—for example, to track both the hand and the head of a user in VR with an HMD. The Polhemus company continues to sell related sensing products today.

12.4.3 Early Tracking of the User's Fingers—Gloves

Although Put-That-There used the direction of the user's arm to indicate where the user was pointing [Bolt 1980], it is much better to be able to tell if the user is making a pointing pose with their fingers and which way the finger is pointing. Therefore, techniques for sensing the user's hand pose and gestures in the air were investigated early on.

One of the first approaches, starting around 1982, was to use a glove with embedded flex sensors to measure the bending of each finger (see Figure 12.8). The most famous version was by Jaron Lanier and Tom Zimmerman from the VPL company, and it was called the *Data Glove*. It used a Polhemus sensor mounted on the back of the hand at the wrist for 6D positioning [Zimmerman et al. 1986]. Some models provided *piezoceramic benders* at each finger that vibrated silently as a form of tactile feedback [Zimmerman et al. 1986]. Many of the 3D interaction techniques investigated during that period used Data Gloves, but they were fairly expensive.

A cheaper version was created by Mattel, with help from Lanier and Zimmerman, and marketed by Nintendo starting in 1989 as the *Power Glove*. Although only two games were developed for the Power Glove, and it was not a successful commercial product, it did inspire researchers such as Randy Pausch to investigate inexpensive approaches to providing and interacting in VR, with his famous 1991 paper "Virtual reality on five dollars a day" [Pausch 1991].

However, wearing a glove is awkward, especially since people have different hand sizes, it only works on one specific hand, it is difficult to get high accuracy,

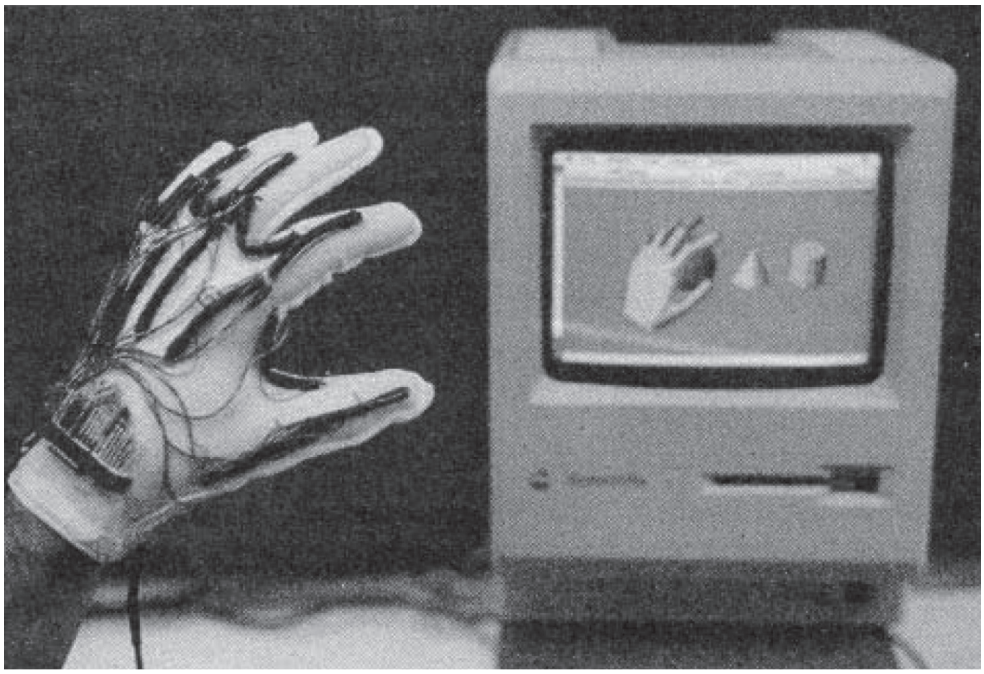


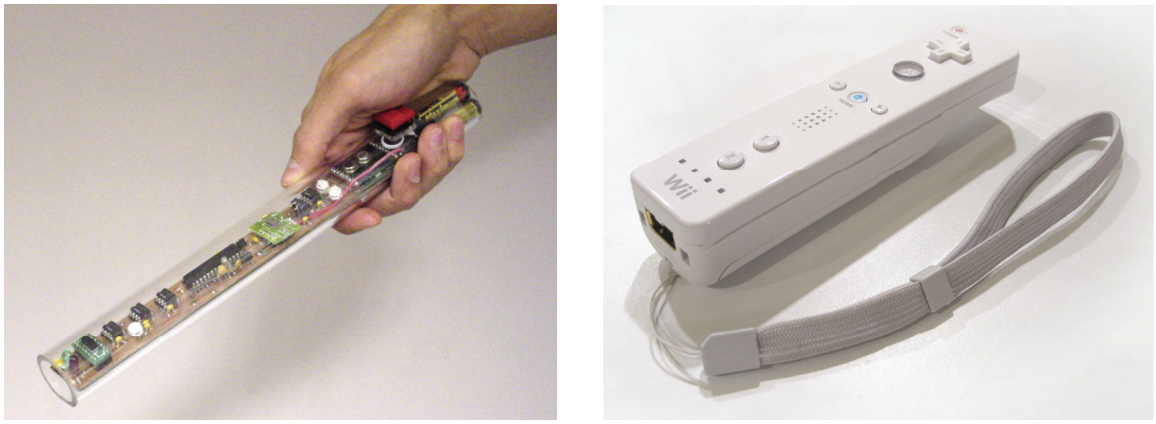
Figure 12.8 Data glove with outer covering removed to show sensors. Source: Fig. 3 of Zimmerman et al. [1986].

and there is a cord dangling off it—so a key goal has been developing other approaches.

12.4.4 Tracking a Wand

Rather than wearing a tracker strapped to one's wrist or a special glove, what about holding a wand? A system from 1997 investigated a digital conductor's baton that could control electronic music and recognize some gestures in the air [Marrin 1997], but it required a cord. In 2003, Andy Wilson at Microsoft Research created the *XWand*, which incorporated a wide range of sensors and was cordless [Wilson and Shafer 2003]. It contained an accelerometer, a magnetometer and a gyroscope for measuring rotations, a button, and an infrared (IR) LED that was viewed by two fixed cameras for positioning in 3D (see Figure 12.9(a)). Various interaction techniques were explored, including pointing to lights in a room to turn them on or off, pointing to a stereo and rolling the wand to control the volume, or controlling a computer cursor by pointing toward a monitor. As discussed in Sections 4.11 and 4.14, cursor control with these devices is inaccurate and tiring, and the other envisioned uses have not come to fruition.

However, the idea of using a wand has been influential in the gaming world. The Nintendo Wii from 2006 was shipped with a wand as its main controller (see Figure 12.9(b)). This was a revolutionary change since competing game consoles used conventional controllers like Figure 4.8(b). The Wii controller sensed rotation and acceleration and could detect its 3D position in space using an IR camera



(a) (b)

Figure 12.9 (a) XWand. (b) Nintendo Wii Remote. Sources: (a) Fig. 1 of Wilson and Shafer [2003]. (b) By Greyson Orlando, Public Domain, https://commons.wikimedia.org/w/index.php?curid=1447832.

that sensed the signal from IR LEDs in a bar that was put on top of the television. This enabled a completely new class of interactions where the users made various motions and pointed.

12.4.5 Camera-Based Techniques

Beyond the above devices, it is often desirable to accurately track a person, preferably avoiding the user needing to hold or wear anything special, and especially avoiding wires to the user's body.

High-end camera-based systems, which often track the entire body, not just a hand, are called *motion capture* and are frequently used to provide realistic motion for special effects creatures in movies and video games. These date back at least to the 1970s for computer-based techniques [Moeslund and Granum 2001], with manual techniques (such as measuring the gait of horses using cameras) dating back at least to 1887 [Barrey 1999]. The first commercial system was probably the Vicon camera-based tracking system dating from the early 1980s, which used multiple fixed cameras to provide 6DOF information at millimeter precision (and continues to be a major player in this space, www.vicon.com). However, these systems often require the user to wear reflective dots or balls so that the cameras can more easily track points on the body.

Motion capture systems are rarely used for interaction techniques beyond tracking the motion itself. Some counterexamples include systems that try to recognize gestures in the air (see Section 14.1) and try to interpret sign language to or from the deaf using cameras (see Section 14.3.1.1). For example, studies of interaction techniques with a handheld projector used reflective balls mounted onto the projector (see Figure 12.10(b)), which were tracked by a Vicon system [Cao et al. 2007] (see also Section 13.6). A fictional example was shown in the popular movie *Minority Report* from 2002, which featured a character played by Tom Cruise putting on special gloves with blue lights on three fingers of each hand to make gestures in the air to control a large display. The realism of the tracking and gestures was attributed to having John Underkoffler, then from the MIT Media Lab, as the "science and technology advisor for the film" [Clarke 2002].

The field of camera-based body tracking without needing to hold a wand or wear anything special was revolutionized by the release of the Microsoft Xbox Kinect in 2010 [Zhang 2012]. This game accessory included IR projectors and detectors that created 3D maps of the objects in front of it, plus an RGB camera to add color (Figure 12.10(a)). The Kinect has relatively low-resolution, so it was mainly used to measure whole body and limbs poses and movements (rather than, e.g., finger poses or motions), but it did support tracking multiple people. In addition to its use in games for the Xbox, the Kinect was widely used by researchers to study body and

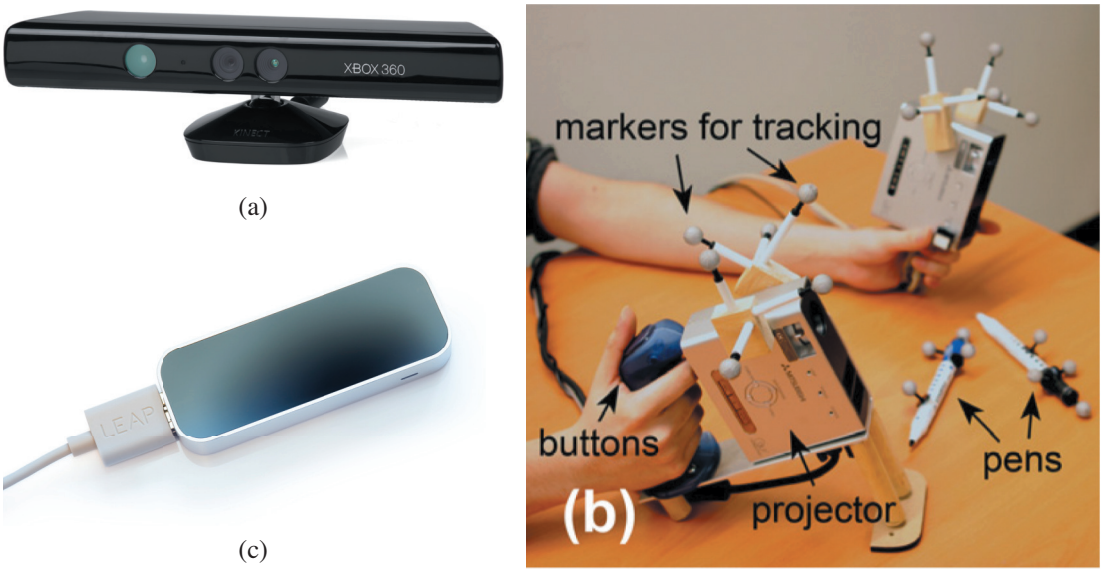


Figure 12.10 (a) Kinect for Xbox 360. (b) A handheld projector with attached balls as markers for tracking. (c) Leap Motion sensor. Sources: (a) By Evan-Amos, Public Domain, https://commons.wikimedia.org/w/index.php?curid=16059165. (b) Fig. 1b of Cao et al. [2007]. (c) By SkywalkerPL, CC BY 4.0, https://commons.wikimedia.org/w/index.php?curid=48288498.

limb tracking in the air for a wide variety of kinds of projects. Microsoft released an official SDK (software development kit) in 2012, which accelerated interest and uses [Zhang 2012].

Another camera-based sensor is the Leap Motion (Figure 12.10(c)), released in 2012, which was designed to be put on a desk looking up, typically in front of the keyboard. The Leap Motion is higher-resolution than the Kinect, but it only has about a one m^2 sensing area—so it was primarily designed to recognize the user's finger and hand poses and motions above it. Recently, the Leap sensor has been used attached to an HMD to recognize gestures in front of the user.

12.4.6 Input Techniques with Haptic Feedback

Haptics is the study of touch, so *haptic feedback*, also called *tactile feedback*, is when the computer provides feedback that the user can *feel*. When what the user feels is a force that the user can push against, then this is called *force feedback*. An example of haptic feedback that is *not* force feedback is putting buzzers on the fingertips of a glove (see Section 12.4.3) that the computer turns on and off, so the user can feel

the vibrations but it does not interfere with the user's motions. Although it is possible to have haptic and force feedback with 2D input devices, this is rarely done, so haptics primarily is relevant to 3D input techniques.

The classic way to provide force feedback is with some kind of mechanical linkages so that motors or mechanical stops can be controlled by the computer to push back or stop the user's motions.

One of the first commercial systems to provide this was the PHANTOM device, originally released around 1994 by the SensAble Technologies company based on work with the MIT Artificial Intelligence Laboratory [Massie and Salisbury 1994], and now available from 3dsystems.com. It had a thimble-like cup that fit around the tip of the user's finger (and later versions provided a stylus, see Figure 12.11(a)), which could be moved and measured in 6D (including rotations), and the device could also provide feedback using three DC brushed motors. Under software control, the user could feel like they ran into a wall or the ground, were moving through a viscous material (like sand or cutting skin in a surgical simulation), or that the surface had a texture [Massie and Salisbury 1994]. The devices seem to cost between USD $2,800 up to $13,000 today.

A similar system is the Falcon from Novint, which has a ball at the end of the arms (see Figure 12.11(b)) and provides 3D sensing and haptic feedback (no rotations). The original version was released starting around 2000, and then Novint created a consumer version for games starting around 2009, which sold for around USD $189.00. However, it was not successful, and the company's entire website seems to have disappeared around June of 2015, according to https://web.archive.org/web/20090417023505/http://home.novint.com/products/novint_falcon.php.

(a) (b)

Figure 12.11 (a) Phantom Omni. (b) Novint's Falcon. Source: Fig. 3 of Faeth et al. [2008].

As with other 3D input devices discussed in this chapter, these haptic devices have certain uses such as providing realistic feedback for moving through simulations and in training (e.g., in medicine). However, there are few actual interaction techniques that have been developed using them. The key IxT is to move an effector in a virtual scene and press a button to cause the effect (e.g., to pick up or cut the object at the cursor). Often, a different button switches to using the device to adjust the scene itself or the camera (e.g., panning around).

12.4.7 Other Sensors

There are a wide variety of other sensors that have been used to provide input to computerized devices. Some of these, such as pressure sensors on toys, do not necessarily count as 3D IxTs on their own but are still useful to cover here since they often form part of an interaction with a 3D object.

One early example was the NOOBIE stuffed animal from 1988 by Allison Druin [Druin 1988], which was big enough for children to sit in its lap (see Figure 12.12).

Figure 12.12 NOOBIE stuffed animal with pressure sensors and a display on its stomach. Source: Picture courtesy of Allison Druin, reprinted with permission.

NOOBIE incorporated pressure sensors so that by squeezing various parts a child controlled a noise and what was shown on the screen in NOOBIE's stomach. Nowadays, many toys, including toys for infants, have sensors for squeezing that cause actions such as noises or blinking lights.

An early exploration of other sensors on a handheld mobile device includes Ken Hinckley's influential work at the Microsoft Research Lab in 2000, where he attached a tilt sensor (accelerometer) and an IR proximity sensor to a PDA [Hinckley et al. 2000]. He explored many interaction techniques that used these sensors including "recording memos when the device is held like a cell phone [since it was a PDA with no phone capabilities], switching between portrait and landscape display modes by holding the device in the desired orientation, automatically powering up the device when the user picks it up ... to start using it, and scrolling the display using tilt" [Hinckley et al. 2000]. As discussed in Section 5.11.2, *tilt-to-scroll* is a bad idea, but many of the other interactions were adopted by today's smartphones, starting with the first iPhone in 2007 that included proximity and tilt sensors. Later versions of the iPhone added additional built-in sensors and interaction techniques using them, such as using the accelerometer to detect when the user shakes the device—to trigger an *undo* operation (Chapter 15), a magnetometer in 2009 so the device can detect its orientation, and a GPS antenna, which together support compass and mapping actions (like rotating a map to the correct orientation). Another recent example is the Google Pixel 2 phone from 2017 that has sensors on the sides that are primarily to support *squeeze to talk* [Quinn et al. 2019], see also Chapter 35.

A completely different kind of input sensor is shown in Figure 12.13, which provides an actuator and sensor for each joint so that the puppet will change under computer control, but more importantly for this book, the user can change the joint angles to control the character in the computer [Yoshizaki et al. 2011]. Applications described in the paper include reachability assessment and visibility analysis, such as for the controls for vehicles, as well as the obvious entry and evaluation of postures for virtual characters.

12.5 3D Interaction Techniques

With the wide variety of input devices to provide input for 3D, there are surprisingly few interaction techniques in wide use. Since one of the dominant uses for 3D is for games, interaction techniques for games are covered first, and then interaction techniques using a regular mouse (or other 2D pointing device like a touchpad) and keyboard. Finally, some actual 3D interaction techniques and 3D widgets are covered, but all of these are from research projects rather than commercial systems. Interestingly, 3D IxTs are some of the rare examples of using both hands to control

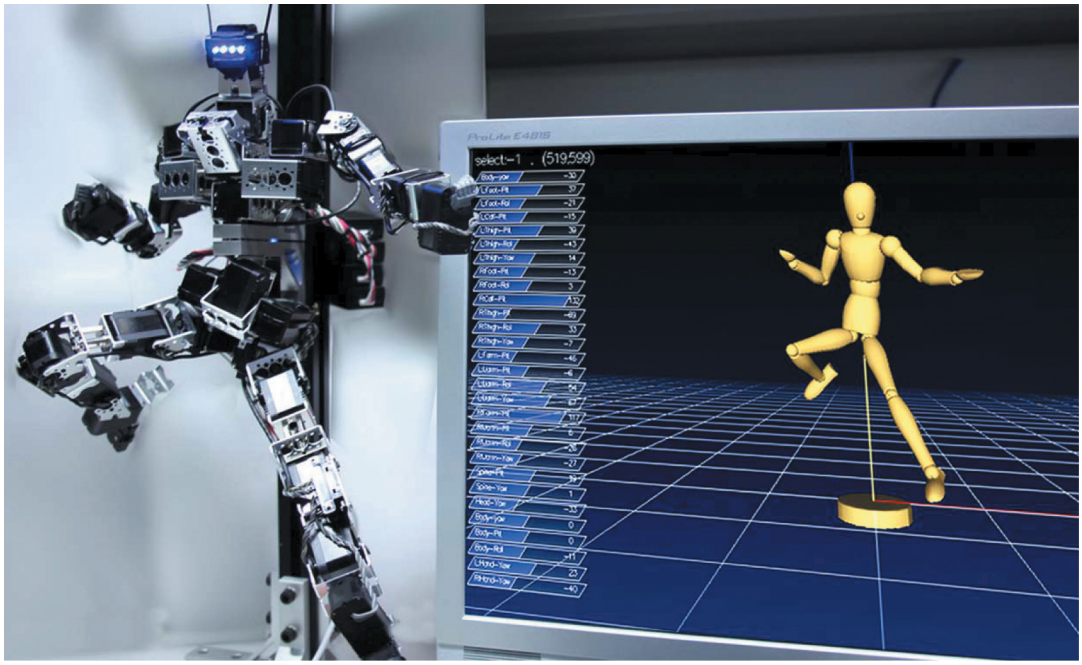


Figure 12.13 Articulated "marionette" with sensors in each joint. Source: Fig. 1 of Yoshizaki et al. [2011].

the interactions because some 3D input devices come in pairs (e.g., Figure 12.1) or use a camera that can see both hands.

12.5.1 IxTs for Games

As already mentioned, the key activities in most games are directing a character's movement, often with arrow buttons or a joystick, and triggering actions, often with buttons, so the interaction techniques are pretty simple. Many games add complexity through combinations of these, such as holding down multiple buttons to affect a jumping movement or holding down buttons for specific lengths of time to control aspects of the resulting behaviors (such as how much the character speeds up). Most games have setup and choice menus that users interact with using conventional means, such as moving around the menu with arrow keys and selecting with a button or pointing at the screen with the Wii controller. These widget-based interactions are covered in the various chapters for those widgets. Interestingly, games have provided an opportunity to explore more non-standard interaction techniques such as pie menus (Section 7.3.9.1 and Chapter 33) and gestures. They also are one of the few places with true two-handed interactions, such

as holding something with one hand while interacting with it with the other hand (see also Chapter 25).

The original arcade games (which were later brought home to consoles and PCs) only supported 1D or 2D motion. For example, in the original *Pong* game from Atari in 1972, each user has a knob to turn to move the paddle up and down (Figure 12.14(a)), and in *Pac-Man* from Namco in 1980, the user had a joystick that could be moved in four directions (Figure 12.14(b)). *Centipede* from Atari in 1981 used a trackball (Figure 12.14(c)). Even *Asteroids* (Atari, 1979), although it would seem to be 3D since it was set in space, was actually 2D as the user's ship and asteroids would only move in 2D, controlled by turn, thrust, and shoot buttons. Driving arcade games typically use a steering wheel, which is just a knob (sometimes with force feedback), along with 1D sliders as the foot petals. A few games have special purpose inputs, like a shooting game where the gun has a detector for where on the screen it is pointing (and is therefore a 2D pointing device). *Dance Dance Revolution* (Figure 12.14(d)) uses buttons stepped on with the player's feet, which they are required to hit based on the sequence displayed on the screen that is synchronized with the music.

As discussed above, console games are usually controlled by a conventional controller (like Figure 4.8(b)) where the buttons and joystick(s) control the direction of the character, the actions, and sometimes the location of the camera.

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

Figure 12.14 (a) Atari *Pong*. (b) Namco *Pac-Man*. (c) Atari *Centipede*. (d) *Dance Dance Revolution*. Sources: (a) By Chris Rand, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=28364913. (b) Namco, Fair use, https://en.wikipedia.org/w/index.php?curid=27921370. (c) By joho345, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=31079276. (d) Dance Dance Revolution, Fair use, https://en.wikipedia.org/w/index.php?curid=53166905.

In *side-scrolling* games, typified by Nintendo's *Super Mario Bros*. from 1985, the character mainly moves left or right, with the option to jump. The character generally stays in the center of the screen, and the scenery scrolls. The typical interactions among objects are that good objects (like coins) are picked up by running into them, and bad objects are avoided by jumping over them and killed by landing on top of them, although every game adds unique twists to the interactions. Modern games have a more 3D world, where the character can move along the ground in any direction using the joystick, along with jumping and fighting using buttons. Specialized controllers have been released with certain games, such as a floor pad for the home version of *Dance Dance Revolution* for the Sony PlayStation (around 1999) and pretend musical instruments for *Guitar Hero* (around 2005). The user interacts with the Wii controller, as discussed in Section 12.4.4, by moving it in space and pressing buttons.

Other games are played on regular PCs and generally use conventional mouse and keyboard interactions. When individual keyboard keys are used to control behaviors, the game will often turn off auto-repeat so keys can be held down. In fact, some games use the low-level interfaces that allow, for example, the game to use the Shift keys to control flippers, which will stay triggered as long as the corresponding Shift key is held down. It is also common to use some keys on the left of the keyboard, such as WASD or ESDF, to be an extra set of arrow keys, which are in the same configuration as the right-handed arrow keys. Beyond selecting things on the screen, the mouse might be used as a joystick—moving in the different directions mapped to pushing a stick that far in that direction. Finally, one could always buy accessories like a conventional game controller or game-specific devices that plug into a computer, originally into a regular serial port or special "game port," and now into a USB port.

Handheld devices (Chapter 3) have always come with at least one game, but usually the interaction techniques were conventional. For example, the Palm Pilot (Section 3.13) was shipped with a game called *Giraffe* where the user had to enter Graffiti characters as quickly as possible, to help with training. The *Snake* game was common on early phones since it only needed arrow keys. Third-party games were also available for free or purchase.

As the hardware on smartphones became more capable and people spent more time on their phones, the mobile gaming market took off. In fact, as with computing in general (see Figure 3.2), the game market is becoming dominated by mobile games (see Figure 12.15) [Warman 2018]. The interactions for most mobile games involve tapping and swiping across game graphics, but some newer games have taken advantage of the other sensors on the phone. For example, some driving games use the accelerometers, so the user tilts the phone left and right to steer.

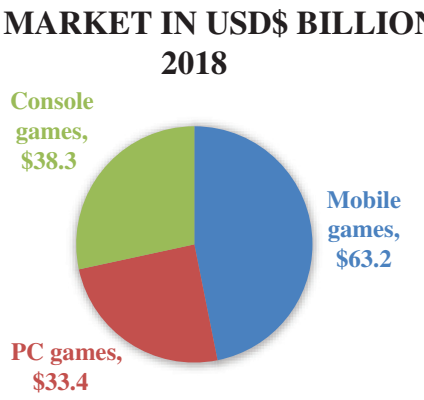


Figure 12.15 Distribution of the US$134.9 billion game industry in 2018. Source: Data from Warman [2018].

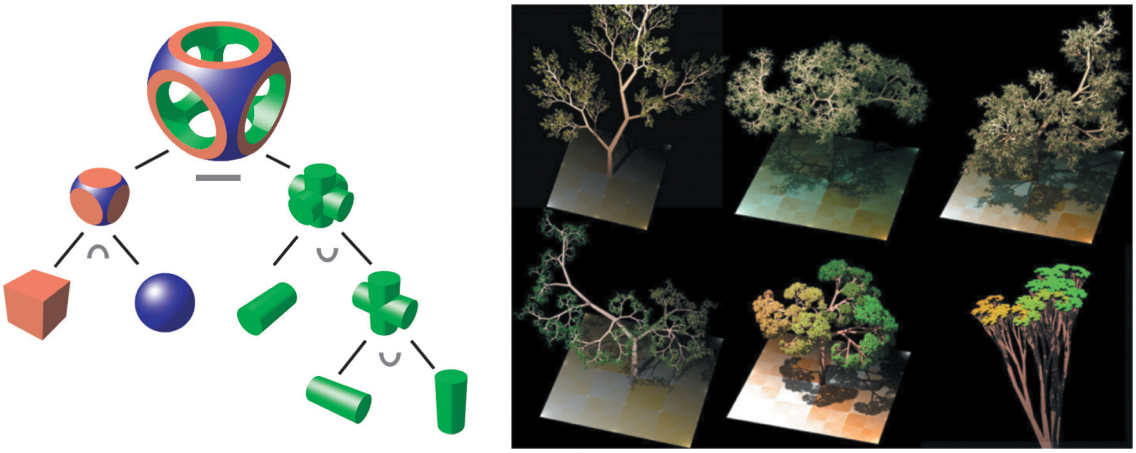
Another class of interactions is for *exertion games*, which are games that have an input mechanism that requires intense physical effort that becomes physically exhausting [Mueller et al. 2008]. This definition is designed to capture games like *Dance Dance Revolution* (Figure 12.14(d)) and sports games where the user must make large motions with the Wii controller (like pretending to bowl or swing a lightsaber), and to *not* cover games where the only physical effort is in the fingers (e.g., using a regular controller). The interactions in exertion games are generally the exertion itself—that is, the computer measures the gross motor movements and uses these to control the behaviors. Thus, the speed of running, the accuracy of stepping on the *Dance Dance Revolution* arrows, or the path of the arm when "bowling" are the inputs, and the feedback shows the in-game results. These may be measured by a variety of kinds of sensors, as discussed above.

12.5.2 IxTs Using the Mouse

For 3D environments on regular computers that are not games, the primary input device is still the common 2D mouse (and mouse substitutes like touchpads on laptops). The question then becomes how to map the 2D coordinates from the mouse to control up to 6DOF when moving and orienting an object? Even when just selecting an object, it is not obvious how to map the 2D mouse position into a point in 3D space. Further, if one is rotating the scene or an object, one needs to design how to map the 1D scroll wheel or the 2D mouse into the 3D rotations. There are no standards and different applications use different interactions for this, so it is impossible to provide a comprehensive discussion. Therefore, this section provides some examples.

The most common technique for drawing a 3D object is to just map the flat plane of the screen into a plane in the 3D world (such as the ground plane), and then after a 2D shape is drawn on that plane, the lines and curves can be *extruded* into surfaces of a 3D object, typically perpendicularly to the original plane. The drawing on the first plane can then use the same or similar techniques to regular 2D graphics drawing (Chapter 10). This works best when drawing parts with flat surfaces, like parts of machines, buildings, or furniture created in a CAD program (Section 2.5.1.9). A variety of 3D editing operations with the mouse are then supported, such as skewing and stretching, moving, rotating, smoothing corners, and so on. A recent research report from Autodesk explores continuously interpolating the cursor depth by inferring what users intend to select based on the cursor position [Zhou et al. 2022].

Another approach is called *constructive solid geometry* (CSG), where users select 3D shapes from the palette, place them into the scene, and use operations such as adding or subtracting shapes (to make holes). Again, the movement of objects with the mouse is usually along the plane viewed on the screen. Although CSG applies to using simple 3D shapes like cubes and cylinders (see Figure 12.16(a)), modern systems provide complex hierarchical shapes as primitives, such as people, animals, monsters, furniture, and so on. Often the components and parameters of these shapes are controlled with complex dialog boxes.



(a) (b)

Figure 12.16 (a) Constructive Solid Geometry (CSG) creates complex shapes through union and intersection of basic shapes. (b) Realistic trees created by procedural generation, controlled by changing parameters and a random number. Sources: (a) Zottie, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=263170. (b) Solkoll, Public Domain, https://commons.wikimedia.org/w/index.php?curid=61990.

A third approach is to use *procedural generation*, where an algorithm will generate objects such as trees, mountains, smoke, or fire based on a mathematical model such as *fractals*, *particle systems*, or a physical simulation (see Figure 12.16(b)). Again, these are usually controlled by parameters typically set using dialog boxes.

In all of these approaches, the user will frequently pan, zoom, and rotate the scene to see the emerging shapes from different angles and control along which plane the mouse operates. A common method is based on the 2D hand scrolling (Section 5.7), with which the user can pan around the scene. A different mode allows rotation, often from a fixed origin somewhere in the scene, and then moving in 2D rotates the scene left, right, up, or down in 3D around that point. The interactions for controlling the view position and rotation have evolved through significant research dating back to the 1980s (e.g., Chen et al. [1988]) and through large-scale deployments in 3D editing software.

A variant on a single-view editor is to provide four views each in its own pane, where one pane is a perspective view and the other three show the object along each of the axes, with the origin typically at the center of the object (Figure 12.17). The cursor position and all edits are shown simultaneously in all views.

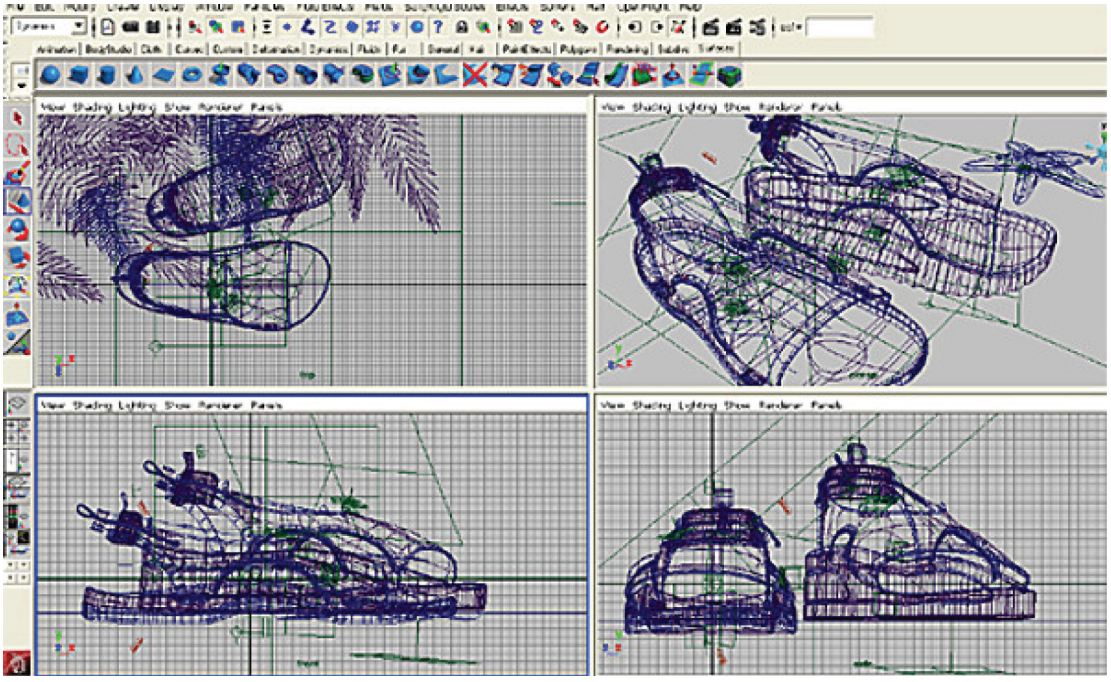


Figure 12.17 Four views of a 3D object—along each of three axes and perspective.

A completely different approach was investigated by Takeo Igarashi in his Teddy program from 1999, where the user sketches curves on a tablet and the system uses heuristics to guess the desired 3D shape based on the drawing [Igarashi et al. 1999]. It uses the context of the existing drawing to help determine the shape and roles of the new curves. For example, drawing a circle on a shape and then a curve from and back to that circle will create a cylinder as an arm or leg (see Figure 12.18). Teddy provides a variety of editing interaction techniques such as drawing a line across an object to cut it, drawing along an edge to edit it to be slightly further in or out, scribbling over a line to delete it, and so on [Igarashi et al. 1999].

12.5.3 IxTs Using 3D Inputs

Although a mouse is ubiquitous and comfortable to use, it makes sense to investigate 3D or 6D input techniques for editing in 3D space. For example, a key use for the PHANTOM 6D device (Section 12.4.6) is drawing, editing, and manipulating 3D models. Typically, the interaction techniques are straightforward mappings of equivalent 2D techniques.

Other devices are used primarily when standing up, often in a VR or AR environment, such as a data glove (Section 12.4.3) or wand (Section 12.4.4). Some more

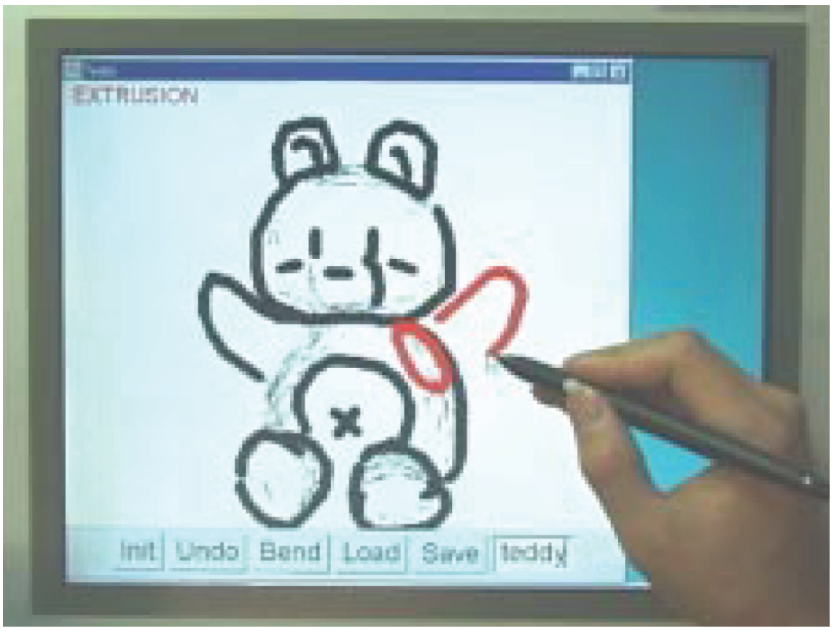
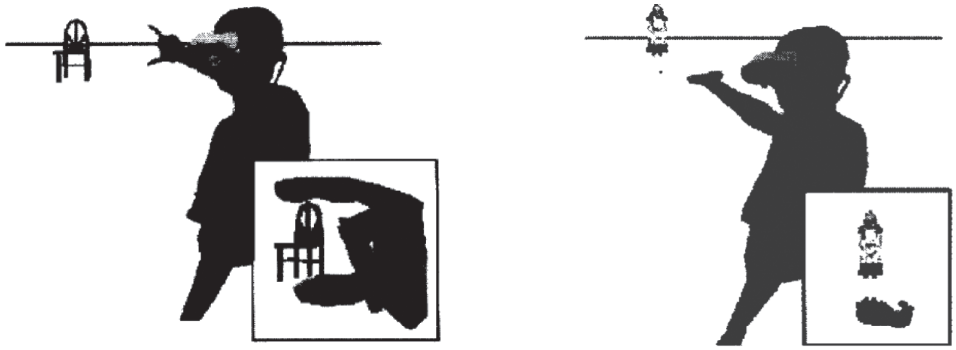


Figure 12.18 In Teddy, the user draws fluid curves to define 3D shapes. Source: Fig. 1 of Igarashi et al. [1999].

innovative interaction techniques have been investigated for these situations. For example, a problem in VR is that objects of interest may be visible but out of reach (beyond the reach of the user's hands in the virtual world). An obvious approach is to cast a ray (like a laser pointer ray) from the user's finger pointing direction and select the first object that the ray intersects. Some systems have the ray come out of the user's wrist or fist for stability and ease of sensing. Another approach is to let the user surround the object in the user's view (called the *head-crusher technique*, Figure 12.19(a)) or just look like it is in the user's palm (Figure 12.19(b)) [Pierce et al. 1997]. A variety of other interaction techniques have also been tried, but there are no standards or consistency across all VR systems.

As mentioned above, for 3D or 6D sensors such as wands (like the Wii controller) and cameras (like the Kinect or Leap Motion), the interaction techniques are pretty rudimentary. Generally, the user can point to objects (for the Wii, only on the screen) and use a button to select that object. The Leap Motion can recognize poses of the hand, but the original Kinect was too low-resolution to see the user's fingers and was limited to body poses [Zhang 2012]. Later versions had somewhat better resolution, but still there are few standard or consistent interaction techniques across applications or devices.

Other considerations for input devices with respect to interaction techniques are their accuracy, resolution, and fatigue. As just mentioned, some sensors are low-resolution. Further, the sensors themselves have quite a bit of jitter, so even if the user is entirely still, the reported values will change. But since most of these require the user to hold their arm and hand out in the air, users are not able to hold



(a) (b)

Figure 12.19 3D manipulations for a VR environment with tracked hand and fingers. (a) A third person view of the Head Crusher technique. The inset shows the first person view. (b) A third person point of view of the Lifting Palm technique. The inset shows the first person view. Source: Figs. 3 and 5 of Pierce et al. [1997].

still, introducing more noise into the data (see Figure 4.19). Finally, any device that requires the user to hold their hands and/or arms up will be tiring for extended use (see the *gorilla arm syndrome* discussed for a lightpen, Section 4.4).

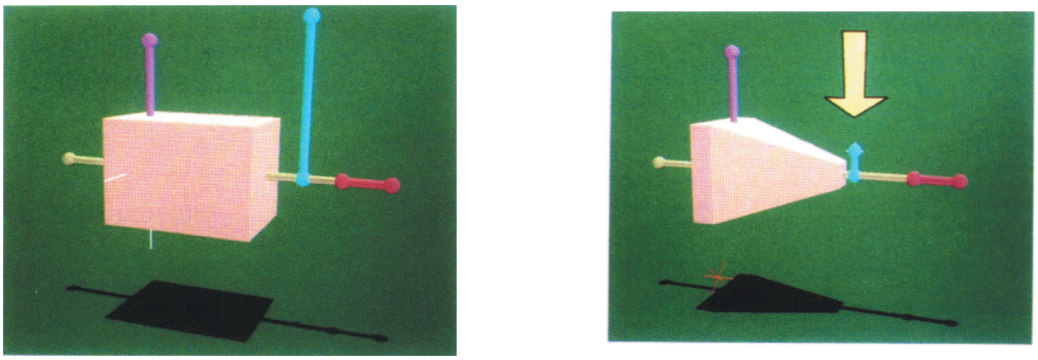
12.5.4 3D Widgets

It is not unusual to see the regular 2D widgets discussed in previous chapters being used in 3D interfaces. For example, 3D editors may have scrollbars and will have lots of buttons, text fields, sliders, palettes, and dialog boxes for the many parameters. It is also not unusual to see a flat menu hovering in space in VR and AR scenes, where the user must select an item by pointing at it.

But what about inherently 3D widgets? The main ones I am aware of are *handles* for manipulating 3D objects. Early research [Snibbe et al. 1992] looked at providing handles on simple objects to support a range of manipulations such as translate, rotate, twist, taper, and bend (Figure 12.20). Other research has looked at how widgets can be adapted for use by multiple fingers on touchscreens like tablets or smartphones [Cohé et al. 2011]. A recent version of SketchUp 3D uses conventional handles at the corners of selected graphical objects when changing their size, but they are little cubes instead of squares (see Figure 12.21).

12.6 Summary of Considerations for 3D Interactions

The first consideration for 3D interaction techniques is whether the target is a game console with a game controller, a regular computer with a mouse, a mobile device,



(a) (b)

Figure 12.20 Handles for 3D manipulations of simple objects. (a) The starting configuration for a pink cube and a rack with twist (purple), taper (blue), and bend (red) handles. (b) The taper handle is translated downward, tapering the cube. The deformation range is the region between the twist and taper handles. Source: Figs. 1 and 2 of Snibbe et al. [1992].

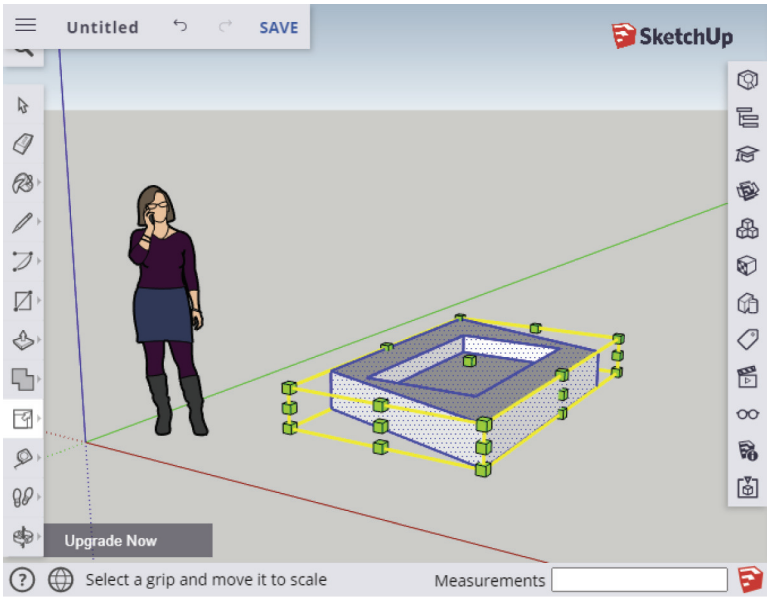


Figure 12.21 SketchUp 3D handles for scaling. Source: SketchUp 3D screen capture on June 23, 2020.

or a VR/AR experience. The kinds of interactions are highly constrained by the input sensors that are available.

For games consoles, most games make do with a conventional game controller with buttons and joysticks, and most games try to be somewhat consistent with other games—for example, to decide how a joystick controls a character's motion or which button causes shooting. Some games do try to be innovative with the mappings to provide new functionality or a more unique experience to the user. There might be alternative controllers, like a Wii wand or Kinect camera, or specialized hardware for arcade games, which open up the design space of interactions.

For a regular computer, the next level decision is the sophistication and duration of use. For example, the Maya program is designed for a highly trained designer who will use the program for long periods of time to create elaborate models, so the program innovates and provides novel interaction techniques that users must learn but which can make them more productive. On the other hand, a program used less often would need more standard interactions, probably more similar to familiar 2D interactions (e.g., possibly using scrollbars and selection handles).

For a mobile device, the designer has the option of using its extra sensors, like for tilting, as well as the touch-sensitive screen for gestures like swiping or two-finger zoom and pan.

Finally, for VR/AR/XR devices, which might include wearing special goggles (Figures 12.1 and 12.2), the designer must consider not only what input sensors are available but also that the user might not be able to see their hands, and that any widgets should not block too much of the user's view of the scene.

Across many devices, there are additional considerations:

- Most 3D sensors are relatively low-resolution (compared to a mouse or touchscreen) and also suffer from jitter, both due to the sensor and due to the user's body parts not staying still.
- If the user is pointing or gesturing in the air, in addition to the lowered accuracy there is also the issue of fatigue, so it cannot be expected to be used for long periods.
- Many of the 3D input and output devices are expensive, limiting their diffusion.
- If haptic feedback is desired, then that limits the options for devices significantly.

Finally, as emphasized throughout this chapter, there is little standardization for interaction techniques in 3D, so the designer will need to do their own survey of similar use cases to see what is currently common in the domain, and they may also end up needing to innovate on new interaction techniques for their specific use cases.

12.7 Chapter Conclusion

Interaction techniques for 3D objects are some of the most volatile and inconsistent, with continuous innovation and research. In fact, in addition to the usual places to publish interaction techniques like ACM UIST and CHI (Section 1.3.6) and the main graphics conference, ACM SIGGRAPH, there are two conferences dedicated to 3D interactions:

- The ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D), operating since 1986, https://i3dsymposium.org/.
- The ACM Spatial User Interaction (SUI) conferences, operating since 2013, https://sui.acm.org/.

Three motivations for the continued innovation and investigation of new interactions are (1) the difficulties of using existing interaction techniques to control complex 3D objects in complex 3D scenes; (2) the continuous release of new games, and the need for them to try to differentiate from existing games, sometimes

through their interactions; and (3) the continuous improvements in graphics and character rendering, increasing the complexity of what needs to be designed and controlled. Since games, special effects for movies, and 3D modeling for manufacturing are all such big businesses, it is not surprising that designers are innovating to make new 3D interaction techniques more efficient and engaging.

CHAPTER 13

Ubiquitous Computing, Physical Widgets, Sensors, Wearables, and Internet of Things (IoT)

13.1 Introduction

Most of the previous chapters discussed interaction techniques that are on screens and interacted with using a mouse or touchscreen or a remote controller. In contrast, this chapter discusses interaction techniques for physical widgets. Although there is some overlap, especially for mobile and wearable devices, this chapter focuses on when physical things *themselves* are made to be interactive, generally through embedding sensors, actuators, and other computerized devices in them.

13.2 Definitions

Here are some more formal definitions of the concepts and terms relevant to this chapter.

- *Ubiquitous computing* (*UbiComp*), also called *pervasive computing*, was defined in Section 3.4 as when computation is available anytime and anywhere, in a manner where computing devices "weave themselves into the fabric of everyday life until they are indistinguishable from it" and become "invisible" [Weiser 1991]. The term is now also used for any computing away from conventional computers, including smartphones and smartwatches, which this book covers in other chapters.
- *Ambient intelligence* (which seems to be mostly used in Europe) refers to when the environment is instrumented so it is sensitive and responsive to people.
- *Information appliance* [Bergman 2000] was an older term referring to PDAs and smartphones, but it is no longer widely used. The concept, as articulated

by Don Norman [1998], was that, like other household appliances like toasters, the information appliance would have a specialized range of functions centered around information. In contrast, today's smartphones, like regular PCs, are multi-purpose and can be used for virtually any information-related task.

- *Context-aware computing* [Moran and Dourish 2001] refers to computing devices that are aware of their physical and digital surroundings. The most common example is a smartphone that knows its location through GPS, but it could also include devices that are aware of the temperature, the ambient light, the time of day, the user's calendar (e.g., to be aware of whether the user is now in a meeting), the user's heart rate (e.g., via a smartwatch), and so on.
- *Tangible user interfaces* (TUIs) (which used to be called *graspable UIs*) are when the user interacts with digital information through the physical environment. TUIs have their own conference series called the ACM International Conference on Tangible, Embedded, and Embodied Interaction (TEI 2020) described at tei.acm.org, which started in 2007. Probably the most influential researcher in the TUI area is Hiroshi Ishii from the MIT Media Lab. As the director of the "Tangible Media Group," his "Tangible bits" paper from 1997 helped define and name the field [Ishii and Ullmer 1997].
- *Tabletop user interfaces* are interactions where the display surface is horizontal. Often these will use conventional or gestural IxTs as covered throughout this book, but because the surface is horizontal, users can place physical objects on the display, which the computer can sense. Then, the user can interact with the displayed image through manipulations of the physical objects. (Theoretically, the same could be done with vertical displays if the physical objects had magnets or were sticky, but this is less common.) ACM ISS: the International Conference on Interactive Surfaces and Spaces (iss.acm.org; formerly ACM ITS: the International Conference on Interactive Tabletops and Surfaces) started in 2006 and publishes research in this area.
- *Physical widgets* or *phidgets*, as defined by Greenberg and Fitchett [2001], "are to physical user interfaces what widgets are to graphical user interfaces. Similar to widgets, phidgets abstract and package input and output devices: they hide implementation and construction details, they expose functionality through a well-defined API, and they have an (optional) on-screen interactive interface for displaying and controlling device state." Intuitively, they are physical knobs, sliders, sensors, and actuators that operate in the real world,

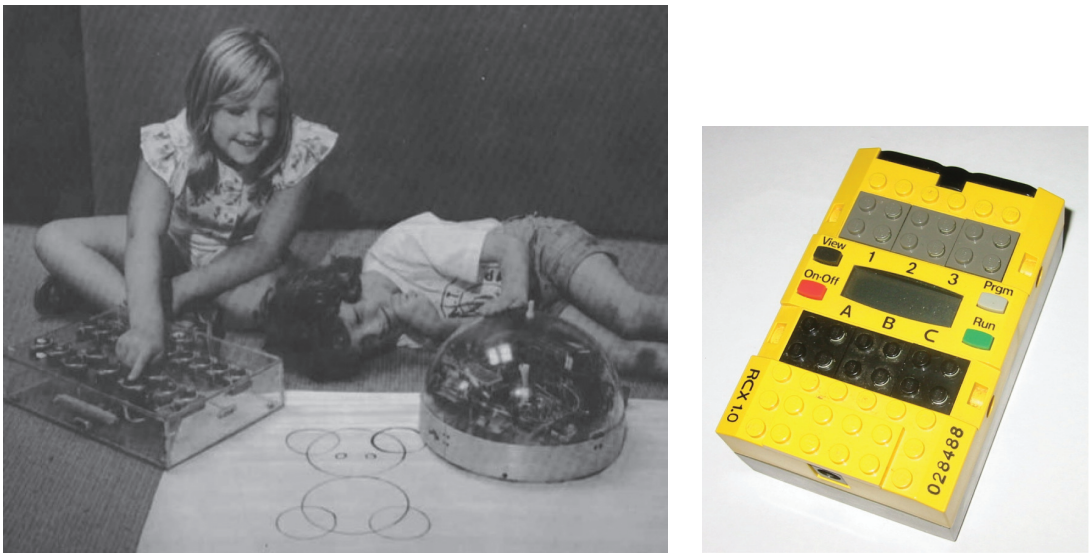
not on a screen, whereas the effect of using them might control on-computer or physical objects.

- *Wearable devices*, also called *wearable computing*, *wearable technology*, or just *wearables*, are simply defined as technology that can be worn on a person's body. Obvious examples include watches and head-mounted displays (HMDs), but this category also includes technology sewn into clothing. The ACM International Symposium on Wearable Computers (www.iswc.net) started in 1997 so there is a lot of research on sensors and interactions that people can wear.
- *Ambient displays*, as defined by Mankoff et al., are "abstract and aesthetic peripheral displays portraying non-critical information on the periphery of a user's attention" [Mankoff et al. 2003]. Since they are designed to *not* be the focus of attention, they generally do not involve any interaction. However, we include them here for the same reason as progress bars (Chapter 6)—they are an important class of interfaces; here, ubiquitous computing applications. A research example was the "BusMobile" that passively showed how close various buses were to a user [Mankoff et al. 2003].
- *Internet of Things* (IoT) refers to a variety of electronic devices that can communicate with each other and the broader Internet, often without requiring any user interaction. The devices are often conventional consumer electronics such as refrigerators and washing machines that have been given the ability to communicate with the Internet, but the term also refers to devices like cameras (e.g., the Ring doorbell, www.ring.com, or a baby monitor) and thermostats (like the Google Nest, www.nest.com) that are only useful when connected. There are many conferences focused on IoT research.

13.3 Turtles to Robots

One of the clearest kinds of computerized physical devices are robots that are controlled by computers. Generally, people interact with these by just writing code, and possibly through screens of conventional widgets, but sometimes more interesting interaction techniques are involved. Here are some examples.

The LOGO programming language, developed by Seymour Papert and colleagues around 1967, had a *turtle* (created in 1970), which was a representation of a very simple form of graphics, used to teach programming [Papert 1980]. Commands to the turtle included turning left or right by an amount, moving forward or backward by an amount, and lowering and raising the pen. Sometimes the turtle was represented on the screen, but there was a physical version (Figure 13.1(a))



(a) (b)

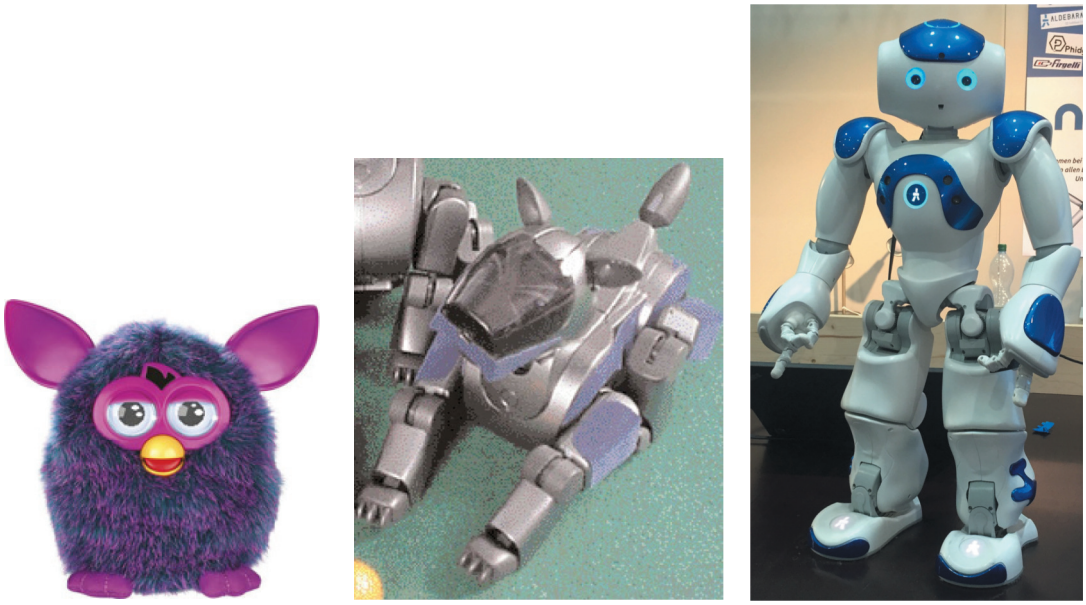
Figure 13.1 (a) LOGO Turtle controlled by buttons. (b) First-generation RCX programmable brick. Sources: (a) Frontispiece of Papert [1980]. (b) CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=513389.

where the pen would draw on the floor as the turtle moved around on wheels. The conventional way to control the turtle was by typing programming commands in the LOGO language, but Figure 13.1(a) shows that there was also a box of buttons that could be used as well.

The Lego company collaborated with Papert's group at MIT and used the "Mindstorms" name from his famous book [Papert 1980] for its line of programmable "bricks," first released in 1998. The controller (shown in Figure 13.1(b)) can be attached to a variety of motors and sensors. The user can write a program in the Mindstorms graphical language or other language (including LOGO), download it into the controller, and then the program can run autonomously. Users would build robots and other structures incorporating the sensors (which included sound, light, and touch) and motors. The resulting structures would then respond as programmed to the inputs to the sensors. The original visual programming language was called *RCX Code* after the name of the original controller (Robotics Command eXplorer, shown in Figure 13.1(b)) and was created using Macromedia Director. The sensors were somewhat low-quality, and the code ran quite slowly on the RCX device, but it provided an important starting point to computing for many children. A later version called NXT, released in 2006, completely replaced

the visual language with a new one called NXT-G, which was based on the professional National Instruments visual language called G that is part of the LabVIEW system. The original RCX language was clunky and hard to edit, and the Macromedia company was purchased by Adobe in 2005, and the Director product was being replaced by Flash which ran better on the Web, so Lego abandoned that environment in favor of the proven LabVIEW system, which has been in continuous use and development since 1986 [Kodosky et al. 1991].

Many other computerized interactive toys have been developed. The NOOBIE giant stuffed toy, with a variety of touch sensors and a screen, was discussed in Section 12.4.7. The Furby toy from Tiger Electronics and Hasbro in 1998 (Figure 13.2(a)) was very popular for a short time [Wired News 1998]. It had proximity sensors on the front and back to detect when petted or tickled, a sensor inside the mouth to detect when "fed," a microphone to detect sounds and music, a tilt sensor to detect when moved or upside down, and a light sensor in the eyes. It could also detect and communicate with other nearby Furbies using infrared. Motors could move the mouth, eyelids, ears, and make it move in place (e.g., to



(a) (b) (c)

Figure 13.2 (a) Furby from Tiger Electronics. (b) Sony AIBO dog. (c) Aldebaran Robotics Nao. Sources: (a) By Furby (WP:NFCC#4), Fair use, https://en.wikipedia.org/w/index.php?curid=37877795. (b) Fig. 3 of Veloso [2002] (excerpt). (c) By ubahnverleih, CC0, https://commons.wikimedia.org/w/index.php?curid=49912982.

look like dancing), and it had a speaker [Wired News 1998]. Key to its popularity was that it had a wide range of cute built-in behaviors and verbal responses that responded to people's and other Furbies' interactions, which changed over time. Nowadays, providing some level of electronic interactivity in dolls and other toys is commonplace.

At the other end of the spectrum for robot toys are systems like the Sony AIBO dog from 1999 (Figure 13.2(b)), the Sony QRIO from 2003, and the Aldebaran Robotics Nao released in 2008 (Figure 13.2(c)). These tend to cost from $2,500 to $15,000 and, although theoretically available as personal toys, are mainly used by educational institutions to practice programming robots. Sensors include cameras in the eyes, touch at various points, a rangefinder for measuring the distance to objects, a microphone, and accelerometers. Actuators primarily focus on moving the various body parts and a speaker.

Actual robots are often designed to be autonomous, with little or no interaction. Some pre-programmed robots provide some level of human interaction such as sensors to avoid hitting or hurting people or conventional screen-based interaction for parameters. Interaction with autonomous automobiles has received significant research from the HCI community (e.g., Riener [2014], Meschtscherjakov et al. [2016], and Martelaro and Ju [2017]), including multiple workshops and sessions at the annual ACM International Conference on Automotive User Interfaces and Interactive Vehicular Applications (AutomotiveUI). Another way to interact with industrial robots is for end-users to program them, often using programming by example (showing the robot what to do, usually by the user moving the actual robot) or some kind of visual programming language on a screen [Myers 1986a].

13.4 Physical Objects

Interactions with tabletop displays are often conventional touch gestures and widgets as discussed previously, but one new class is when the tabletop interacts with physical objects placed on top of it. One early example was the DigitalDesk from 1991 that had a projector and camera mounted above a user's desk [Wellner 1991]. It would detect pieces of paper on the desk, recognize certain text on it like numbers, and then projected a virtual calculator on the desk with which the user could manipulate the recognized numbers. Another early example is the *Graspable User Interface* which used one or more physical artifacts as controls for on-screen objects [Fitzmaurice et al. 1995], and showed a performance advantage [Fitzmaurice and Buxton 1997].

Other researchers investigated detecting other kinds of physical objects on tabletops. For example, the InfoTable [Rekimoto and Saitoh 1999] supported using a laptop's mouse and keyboard to control objects projected onto the tabletop simply by putting the laptop onto the table and "hyperdragging" documents onto or

from the shared tabletop by dragging across the laptop and tabletop displays. The new interaction techniques here were the detection of devices in proximity and gestures that crossed from one device to the other. The InfoTable also used the orientation of the laptop to determine the appropriate rotation of objects as they were dragged to be shared among multiple participants on the table. A later system from Microsoft Research showed detection of a phone in contact with the tabletop and automatic movement of pictures from the phone to the table when the user swiped them, implemented using Bluetooth [Wilson and Sarin 2007].

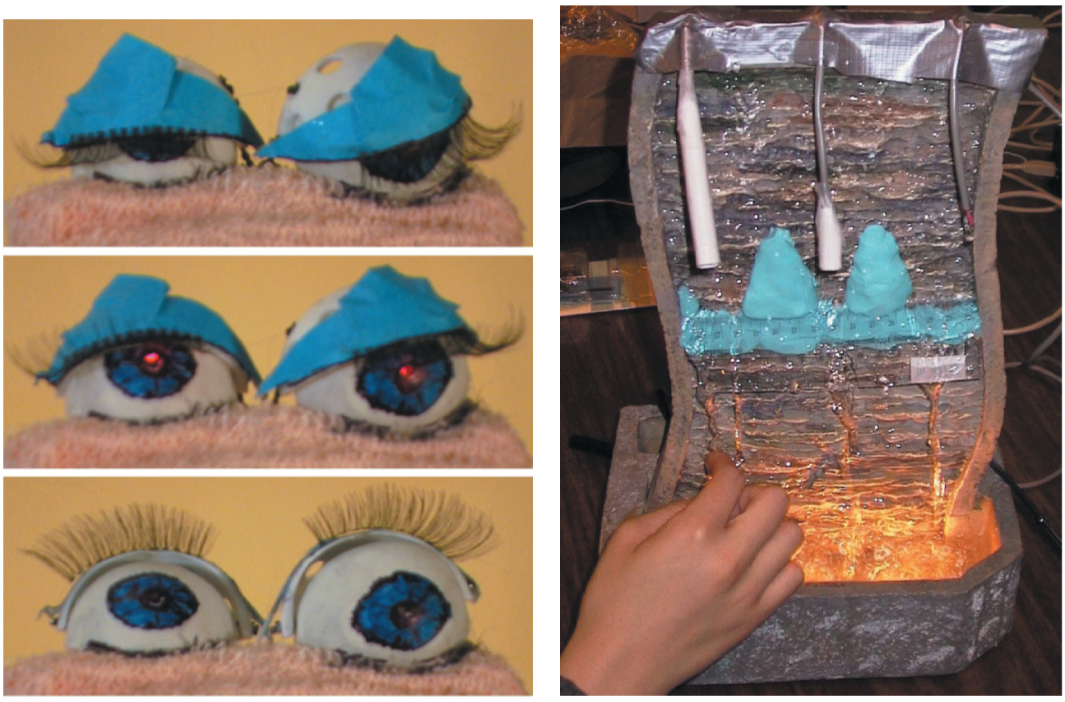
As mentioned above, probably the most active lab researching physical user interfaces is Hiroshi Ishii's *Tangible Media Group* at the MIT Media Lab [Ishii and Ullmer 1997]. They have dozens of examples and projects dating back to the mid-1990s, most of which are interactive in some way, although a few are just ambient displays. However, very few are designed to be reusable across applications, and so may not be useful as interaction techniques, using my definition. One exception is the Urban Planning *URP* system, which provides a tabletop user interface where the locations of pieces of plastic are sensed by the system and control simulations of wind, shadows, reflections, and other effects projected onto the table [Underkoffler and Ishii 1999].

A recent addition to physical objects is the ability to 3D print physical widgets, sometimes connected to whatever they are controlling. For example, a 2016 paper presented 3D printed pneumatic (air pressure) sensors and actuators, including buttons, knobs, and sliders (Figure 13.3) that could connect to electronics or directly to objects that move [Vázquez et al. 2015]. A new paper from Carnegie Mellon University describes devices with internal cables to dynamically reconfigure how the device can move and be moved [Yang et al. 2022].

By 2001, toolkits (Section 2.5.2.1) and interactive tools (Section 2.5.2.2) made programming with *regular* widgets quite easy, but Saul Greenberg's group found that trying to program interactive physical objects with sensors and actuators such as physical widgets was quite hard. It might involve soldering and electrical

Figure 13.3 3D printed pneumatic button (A), knob (B), and slider (C). Source: Fig. 1 of Vázquez et al. [2015].

engineering to wire up the sensors and actuators into circuits, assembly language programming for the low-level controllers connected to those circuits, and communication protocols between those controllers and regular computers. Therefore, they created what they called *Phidgets* [Greenberg and Fitchett 2001], meaning "physical widgets," that provided both the hardware and software infrastructure to make creating objects with Phidgets as easy as with the regular widgets discussed throughout this book. The system provided a variety of sensors and actuators that connected in a standard way to a controller, which communicated to a PC. The programmer could then just write familiar style code on the PC (in Visual Basic in the original version) and the system communicated with the devices automatically. The system provided a simulator so that code could be written even before the hardware was finished, as well as providing debugging support so that the PC could track the status of the components. Students were able to make a variety of ambient and interactive physical artifacts (Figure 13.4).



(a) (b)

Figure 13.4 Phidgets controlling physical objects made by students: (a) Eyes closed, partially open and lit, and fully opened. (b) Waterfall harp that responds to the user's finger position using light sensors. Source: Figs. 11 and 10 of Greenberg and Fitchett [2001].

Now, creating these kinds of devices has gotten much easier with a wide variety of controller boards such as the Arduino (arduino.cc), Raspberry Pi (www.raspberrypi.org), the IOIO-OTG from SparkFun Electronics that is controlled and programmed using an Android smartphone, and many others.

13.5 Wearables

The classic wearable device is a watch, and watches that are computerized have been discussed above (e.g., Figures 3.19 and 8.19) and may use conventional or specialized interaction techniques. Also covered above are HMDs (Figures 12.1 and 12.2), which are another classic type of wearable. HMDs are also often used with other wearable (or handheld) devices for input such as the knob shown in Figure 5.3 or the chorded keyset in Figure 8.11.

Some wearable technology is not interactive, such as the original Fitbit activity tracker armbands that contained a collection of sensors but no display or other way for users to get feedback. Another non-interactive wearable technology would be lights on a shirt that a user does not control.

A commercial example of an interaction technique that is actually part of the clothing is the *Levi's Jacket with Jacquard by Google*, which has sensors built into the cuff that detect gestures such as tapping and swiping and send signals to the user's smartphone—for example, to control music playing [Bohn 2019]. The original version from 2018 used sensors sewn into the sleeve and cost $350 and wore out with washing. A newer version just uses a small fob that is inserted into a special pocket in the sleeve and is removed before washing, and the set costs more like $200.

13.6 Handheld Projectors

An interesting way to interact with the real world is to have the user hold and move around a data projector, and hence move the picture projected from a computer onto various places in the environment. Most early investigations needed to use unwieldy and heavy projectors (Figure 12.10(b)), but now there are relatively bright "pico projectors" using LEDs that are the size of a smartphone that make these more practical. (Projecting a menu onto a person's own body was covered in Section 7.3.10.) Section 12.4.5 above presented one way to track the position of the projector using a fixed camera. In that system, the handle holding the projector had two physical buttons that the user could press, like mouse buttons [Cao et al. 2007].

Some interesting interaction techniques to specifically take advantage of the projected image have been explored. One might think of a whole wall containing an invisible information space, with the projector used to show different parts,

like a flashlight [Beardsley et al. 2005]. This can be extended to multiple projectors (Figure 13.5) with potentially special content showing if multiple projected images overlap at the same place on the wall [Cao et al. 2007]. If the images move with the projector (instead of being fixed on the wall), then each person's projected image might represent their personal information, like their calendar, and overlapping could show a *semantic merge*, like only showing times both people are available in an overlapping image of the two calendars [Cao et al. 2007]. Another idea is to display a cursor at a fixed location in the room, and select based on what picture is moved on top of it [Beardsley et al. 2005] (selecting in this way was introduced at least by 1976 in Bill Buxton's *moving menu* [Buxton 1982]).

13.7 IoT

As defined above, IoT refers to a variety of electronic devices that can communicate with each other and the broader Internet. Often, the devices will be consumer electronics, such as washing machines and thermostats, that would previously have not been connected. Newer devices, such as cameras on doorbells or as part of baby monitors, would also be part of the IoT. When a home has a collection of connected devices, it might be called a *Smart Home*.

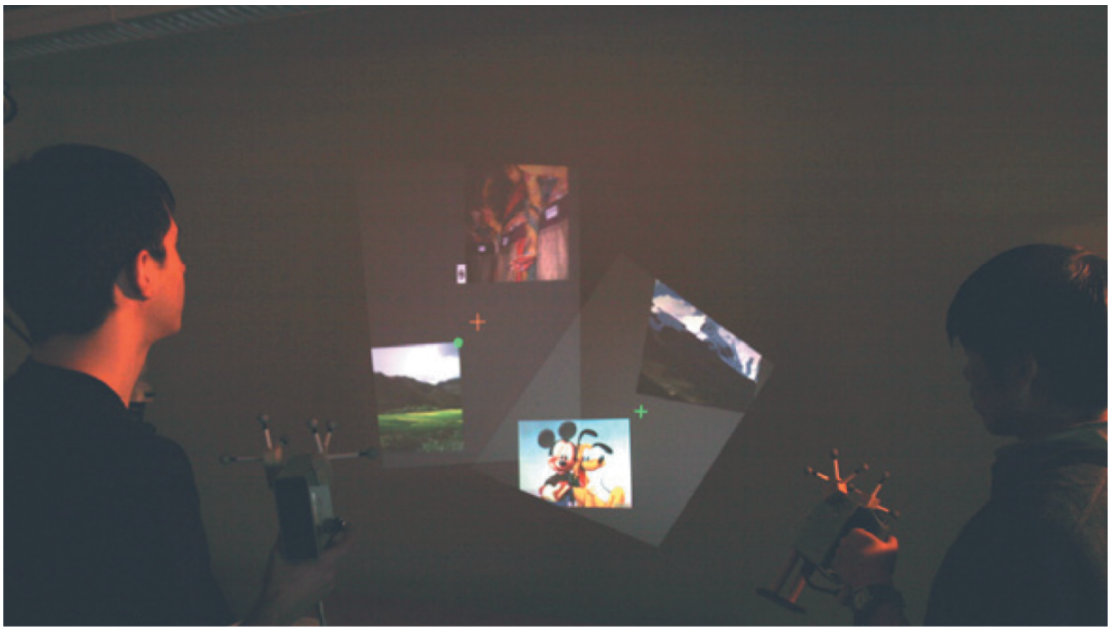


Figure 13.5 Two people each with a handheld projector, exploring an information space on the wall together. Source: Fig. 1c of Cao et al. [2007].

With respect to interactions, some IoT devices have conventional physical or screen-based control panels mostly using buttons and dials, which have been discussed in previous chapters. Other IoT devices are designed to have no user interaction at all. However, a key factor for IoT devices is that they communicate, and therefore can be *remote controlled*, typically from a regular computer, the web, or a mobile device. For example, the Samsung *SmartThings* ecosystem allows lighting, cameras, door locks, home security systems, thermostats, outlets, entertainment systems, and various sensors to be monitored and controlled from a smartphone or voice assistant like Google Home or Amazon Echo (see www.smartthings.com).

Research systems have investigated having the remote control interfaces on mobile devices be automatically generated from a specification of the device's capabilities [Nichols and Myers 2006], and how to allow end-users to create their own customizations and speech interfaces for appliances that only have a smartphone interface [Li et al. 2017]. However, all commercial systems rely on the device manufacturer writing code for each of the supported remote control interfaces. Further, manufacturers use different protocols, which are sometimes proprietary, making it difficult to have devices interoperate. Thus, the user experience often requires interacting with a variety of controls with inconsistent naming and behaviors, for what might seem to be a single task.

13.8 Summary of Considerations

If one was going to build a new physical device with sensors, actuators, and interaction, there are a variety of considerations:

- *Security and privacy*: One of the most important considerations for any physical device in the home or office that can communicate is how secure it is and whether it can be trusted to protect the user's privacy. Experience shows that anything on the Internet or even on a local wireless network is subject to being hacked, which could allow nefarious actors to see through one's baby monitor, listen through the microphone in a toy, turn off the home's security system, and so on.
- *Physical safety*: Clearly, safety is a concern for robots that can move around. Other household or workplace devices that can move must also be careful not to hurt people, even down to avoiding pinching fingers in joints or wheels.
- *Robustness*: Physical actuators and sensors are much more likely to break than screen-based interfaces, especially if embedded in toys or devices in rough or dirty environments.

- *Discoverability*: Although it may be clear to people how to interact with physical buttons, joysticks, and knobs, it can still be difficult for people to figure out how to perform tasks when those controls are overloaded with many operations. For other devices, like a wand, camera-based gestural system, or handheld projector, there is no indication of what one can do with it.
- *Lack of standards*: A related problem is the lack of standards for these kinds of interactions, with each device designer making different decisions, and lots of innovative research every year. This contributes to lower usability and effectiveness for deployed systems.
- *Dizziness and fatigue*: It has been well-established that HMD can cause dizziness in many users, and all controls that require users to have their hands or arms in the air are subject to problems of fatigue. Hence, long-term use of these devices is problematic.
- *Usefulness*: Finally, beyond toys and video games, many of these interactions have not received widespread use, so it remains to be seen whether they will be useful, and in what contexts.

13.9 Chapter Conclusions

As mentioned in Section 13.2, research in the areas of ubiquitous computing, physical widgets, sensors, wearables, and IoT is extremely active, with hundreds of publications every year on these topics. Just as one example, the annual CHI conference has a track called *Interactivity Showcase* where there are around 50 demonstrations, many of which are these kinds of devices. There is also much hype and commercial expectation that these devices will be popular and ubiquitous, and yet, beyond entertainment and commercial robotics, widespread application has been less successful.

CHAPTER 14

Gesture Recognition

14.1 Introduction and Definitions

Gestures and gesture recognition have been mentioned many times already in this book. Section 3.2.5 above defined a *gesture* as "a form of input where the *path* is important to the meaning." Classic examples of gestures are *flick-to-scroll* or drawing a free-hand circle to enter a circle into a drawing package. *Gesture recognition* is the process of recognizing whether the user has entered a gesture, and if so, which one. This is implemented using a *gesture recognition algorithm* in the software. Often, this algorithm will take advantage of various properties of the path, beyond just the locations in space, such as the speed, timing, orientation, size, and so on. The history of gesture recognition is discussed in various sections, including Section 2.5.1.11, throughout Chapter 3, and Sections 8.3.2, 8.3.6.2, 12.4.3, and so on.

By my definition, simply tapping, double-tapping, or long-pressing do *not* count as "gestures" since there is no path (see Section 4.3 for definitions of tap, double-tap, and so on). Further, a normal drag-and-drop does not count either since the path itself is not important—only the start and end points (it doesn't matter what path the user takes from the start to the end, the result is the same). However, current systems must differentiate whether the user taps, double-taps, press-and-holds, or flicks, so a single piece of *gesture recognition software* makes this determination. Therefore, other sources have a more expansive definition of "gestures" in order to include basically all kinds of input from a pointing device.

Another behavior that I do *not* count as gesture recognition is *inking*, which is where the user is drawing freely with a pointing device in a drawing or painting application (see Section 2.5.1.5 for a discussion about the difference), and the *ink* continuously draws wherever the pointer goes. Since the path has no *meaning* when inking, no gesture *recognition* is needed.

Related concepts are handwriting and hand printing recognition, which I *do* count as forms of gesture recognition. These have already been covered in Section 8.3.1, with other gestural techniques for text entry covered in Section 8.3.2.

A gesture will be composed of one or more *strokes*, where a *stroke* is a path made without lifting the pen or finger, or releasing the mouse button. For example, an "o" is usually made with one circular stroke, whereas an "x" is made with two straight strokes. As discussed above in Section 8.3.2, *Unistrokes* [Goldberg and Richardson 1993] and Graffiti (except for entering an "X") are single-stroke gestural alphabets, whereas hand printing requires a multi-stroke recognizer.

Gestures can be performed in the air—for example, to be recognized by a camera or a Data Glove (Figure 12.8), in which case the path to be recognized might be of the hand, arm, or whole body (Section 12.4). A related concept is *pose recognition*, which recognizes when a body part is in a particular configuration, such as detecting when a user's hand is pointing versus making a fist (Section 12.4.3).

A key concept with any of these systems is the recognition *accuracy*, usually measured as the percentage of time the recognizer correctly returns the gesture that the user intends. Some gestures are easy to recognize correctly, such as differentiating the direction of a flick gesture, whereas other gesture sets may be prone to errors due to the similarity of different gestures.

14.2 Advantages and Disadvantages of Gestures

14.2.1 Advantages

As discussed in Section 2.5.1.11, gesture and handwriting recognition has been studied since the 1960s as a more *natural* way to interact with computers. Since people already know how to handwrite and hand print, to draw shapes on paper with a pen, and to use a pen to scratch out or circle items, it seems natural to use the same interactions with computers. However, there are a large number of other advantages to gestures as well.

First, they are very *fast to enter* (*efficient* for the user). Flicking or using two fingers to scroll on a mobile device is much faster than trying to move to a scrollbar and accurately hit or drag on the small targets with a stylus. Similarly, in a drawing program, it is likely to be faster to draw shape outlines and have them recognized rather than having to continuously move back to a palette and select the desired shape and then placing and sizing it.

A related advantage is that a single gesture can *provide both the command and its parameters*. For example, the gesture of drawing a circle can be recognized as the command to create a new circle object, and it also supplies the position and size of the desired circle. Similarly, a scratch-out gesture would indicate which text to delete. This is in contrast with a conventional command that requires separate actions for selecting the command (e.g., the circle icon in a palette) followed

by drawing out the location and size. Dean Rubine coined the term *eager recognition* for when *additional* parameters can be specified as soon as enough of it has been seen to recognize it unambiguously [Rubine 1991a, 1991b]. For example, after drawing an "L" shape to signify a rectangle, the user would get continuous feedback while adjusting the size.

Another advantage is the *large space of potential gestures*, so that all or most of the desired commands can be assigned to gestures.

Finally, from an implementation point of view, conventional mouse-based interfaces are implemented using an event-based model (see Section 20.12.2), and gestural recognition systems can be easily integrated as part of the event model, so programmers do not necessarily need to learn something new to utilize gestures in the user interfaces (UIs) they are implementing [Landay and Myers 1993].

14.2.2 Disadvantages

On the other hand, there are a large number of disadvantages to using gestural interaction techniques, which is one reason they have found only limited use in today's interfaces in spite of being studied since the 1960s [Davis and Ellis 1964].

A key disadvantage of gestures is the *lack of affordance* [Norman 1988, p. 9] so users cannot see from the interface what gestures are supported and where. Unlike menus or palettes that clearly show what is available and how to enter them, typically nothing in the interface shows the user which gestures do what or how to draw the gestures correctly.

A related problem is the issue of *memorability*—the user may not remember which gestures do what even after learning them, especially if there are many gestures.

Although in theory gestures should be natural, there have been cases where the gestures are poorly designed, and hence *not natural*, making them harder to learn. This also exacerbates the issue of memorability. For example, Goldberg's Unistrokes gesture set [Goldberg and Richardson 1993] is hard to learn, motivating the Palm Graffiti gestures to be more like regular letters (Section 8.3.2). This sometimes makes it *difficult to design* a good gesture set.

Another related issue is the difficulty of providing appropriate *feedback* from gestures. If the user makes a delete gesture over some objects, this is a quick way to make them go away, but it is harder for users to know what was deleted since the gesture may accidentally overlap unintended objects. As discussed in Section 10.5, it is usually preferred that users select graphical objects *first* before performing an operation since identifying the desired objects can be tricky. Similarly, feedback about other operations, like creation, moving, or other edits, can be difficult to see, especially if there is a good chance of misrecognition, as covered below.

Since gestures must be recognized, another key problem is *misrecognition*, where the system mistakes one gesture for another. Obviously, this is more of a problem when there are many gestures, but it still happens on current systems with a small gesture set. For example, I am frequently invoking the wrong command on my iPhone, where it misrecognizes a diagonal stroke down from the top (which opens the Control Center) as a vertical stroke from the top (which shows the Notifications screen) or as a vertical stroke from not-quite-the-top (which opens Search). When gestures are misrecognized, the user must typically undo whatever the incorrect gesture did, and then try to perform the correct action either by gesturing more carefully or switching to using a different interaction technique, like a menu command, which can make gesture recognition substantially *slower*.

A related problem is *errors in handwriting or hand printing recognition*, which are notoriously error-prone. As discussed in Section 3.8, the low accuracy of the handwriting recognizer of the Apple Newton was one of the reasons for its lack of success and also was a motivation for the design of Palm's Graffiti gestures to be easier to distinguish (Section 8.3.2).

With respect to text entry, *handwriting is actually substantially slower* than typing or speaking (at about 5 to 23 wpm, with an average of 14 wpm, compared to 35 wpm for typing and 100 wpm for speaking, see Table 8.1). If the gestural text entry method (whether handwriting, hand printing, or other gestural technique) is error-prone, then it will be even slower (and more frustrating) in practice (as parodied in the cartoon in Figure 3.12).

A related issue is that gestures often are *difficult or impossible to perform by people with disabilities* (so they have low *accessibility*, see also Chapter 17). Imagine someone with a hand tremor trying to make a Graffiti or other gesture—that was the motivation for the EdgeWrite gesture set, which uses a plastic square to help stabilize users' movements (Section 8.3.2). And people who do not have any motor control cannot make gestures at all.

Finally, it can be *difficult to implement* a good recognizer for the gesture set. As discussed below (Section 14.4), there are a variety of algorithms available for recognizing the gestures, all of which have tradeoffs, and the algorithm may need to influence which gestures are used (so the gestures are easily and accurately recognized) and/or the implementation may need to take into account the desired gestures, for the same reasons.

14.3 Applications for Gestures

The story of gesture recognition is intimately tied to the story of mobile devices, so much of the history and applications of gestures have been covered in Chapter 3 and elsewhere above. However, since it is distributed throughout various chapters,

it seems useful to have a quick summary of some systems here, approximately in chronological order.

14.3.1 Gestures for Text Entry

See also the quick historical summary in Section 2.5.1.11, and the text entry chapter (Chapter 8) for more about the following:

- The *RAND tablet* from 1964 (Figure 4.11) was used to develop early gestural and hand printing recognition algorithms (see Sections 2.5.1.11, 8.2.4, and 4.9).
- The *Pencept PenPad* (Section 3.5) was launched in 1982 with support for hand printing recognition and other gestures for input [Ward and Blesser 1985].
- The *PenPoint* operating system from Go Corporation, founded in 1987, with the first product released in 1991, also used hand printing (Section 3.5).
- Microsoft supported pen computing starting around 1991 with *Pen Windows*, which provided both hand printing and handwriting recognition (Section 3.7).
- The Apple *Newton MessagePad*, which was started in 1987 and released in 1993, famously touted handwriting as its key text entry technique (Section 3.8).
- *Unistrokes* from Xerox for the handheld *PARC tab* dates from around 1993 [Goldberg and Richardson 1993] (see Sections 3.4 and 8.3.2).
- The *Palm Pilot* was first released in 1996 and supported the *Graffiti* gestural language (Section 3.13).
- *Windows CE*, also from around 1996, supported a gestural language similar to Graffiti, originally called *Jot* (Section 3.14).
- Microsoft released the *TabletPC* in 2002, which provided better handwriting recognition, and also hand printing and a gestural alphabet like Jot or Graffiti (Section 3.16).
- Jacob Wobbrock and I developed a gestural input technique called *EdgeWrite* starting around 2003, which is designed to be easier to enter for people with disabilities (Section 8.3.2).
- The *SHARK/ShapeWriter* research also was first released in 2003, which allows the user to move from key to key on a virtual keyboard without lifting the stylus, thus making shapes (gestures) on the virtual keyboard that are

recognized as words (Sections 8.3.6.2 and Chapter 35). The similar *Swype* technique also dates from around 2003.

14.3.1.1 Recognizing Sign Language Gestures

American Sign Language (ASL), and the equivalents for other natural languages, are a way of making gestures with the arms and hands for communicating with people who are hearing impaired. It is natural to imagine a computer interpreting the signs using a camera, and possibly providing output by showing an animation of an avatar signing. Indeed, there is a great deal of research on this topic, with widely varying success. An early system used a camera on a desk or on a cap pointing down [Starner et al. 1998], reporting a 97% accuracy with an unrestricted grammar but based on only a 40-word lexicon. Other systems have used the Microsoft Kinect [Yang 2015] or the Leap Motion sensors described in Section 12.4.5. Another study reported that when watching an animation of an avatar signing in ASL, there were significant differences in subjective and objective scores for different subpopulations, so that just reporting one overall score can be misleading [Kacorri et al. 2017]. Typical speeds for signing are between 90 and 142 wpm [Al-khazraji et al. 2018], which is faster than normal typing speeds (35–80 wpm, see Table 8.1), making this a reasonable interaction technique for text entry if sufficiently accurate.

14.3.2 Gestures for Proofreading and Editing Text

Besides using gestures for text *entry*, probably the earliest application was for text *editing*, by recognizing *proofreading symbols* drawn on top of manuscripts. Proofreading symbols date back to the beginning of printing, with a specific set of marks generally used (see Figure 14.1). Some of the marks have even been standardized by ISO 5776 "Graphic technology—Symbols for text correction" starting around 1983. An early paper from 1969 studied having a computer recognize these kinds of marks when made over a printed document [Coleman 1969]. With the rise of pen computing in the 1980s (see, e.g., Section 3.5), there was increased interest and research, such as a system from IBM that used gestures made with a stylus on a spreadsheet on a touchscreen to edit the contents [Rhyne 1987]. The Pen computers, like PenPoint from Go, incorporated a number of these editing gestures, such as to insert between characters or draw an "x" to delete [Go Corporation 1992]. However, given that editing text on a computer with regular commands (Section 8.6) is quite quick and easy, there is less need now for gestural editing.

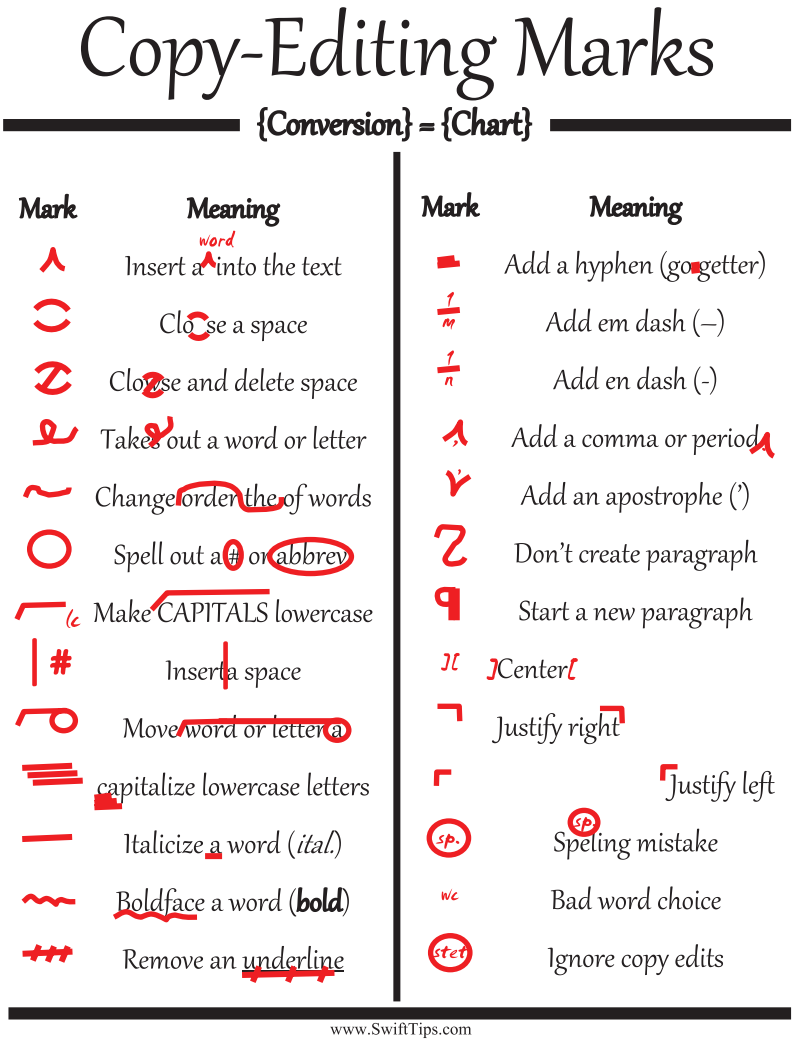


Figure 14.1 Copy-editing marks for proofreading. Source: Reprinted with permission from https://www.swifttips.com/preview/Copy_Edits.

14.3.3 Gestures for Object Entry and Editing

Beyond text entry and editing, gestures have been widely explored for entering and editing other graphics. Possibly the first example was an integrated circuit layout system from Lincoln Labs on the TX-2 from around 1969, which allowed circuit elements to be created, selected, and edited using gestures. This system transitioned into a commercial company called Applicon, arguably the first company to

use gesture recognition in a product [Buxton et al. 2005]. An important feature of the Applicon CAD system was that the gesture recognizer was *trainable*, so the user could define new gestures for commands. It also used a tablet and stylus.

As discussed in Section 2.5.1.11, there were explorations in the late 1970s and early 1980s of using gestures for entering and editing music [Buxton et al. 1979], for graphical objects like circles and triangles [Buxton et al. 1983a], and even for cartoon characters [Krueger et al. 1985]. Resizing graphics by pinching with two fingers was shown in Dean Rubine's 1991 research [Rubine 1991a]. The recognition of sketched or hand-drawn circles, rectangles, lines, and other simple shapes has been a feature of applications on pen- or finger-based systems since the beginning, and continuing with drawing apps for today's tablets, including iPad apps such as ProCreate, GoodNotes, Notability, OneNote for iPad, and even Apple's original Notes app. Often, converting hand-drawn shapes to their exact equivalents is called *neatening* of the drawings. The position and sizes may also *snap* to other objects or grids, as discussed in Section 10.6.6, to further neaten the result.

Another class of gesture recognition systems recognized gestures as graphical objects, but then recognized groups of graphical objects as higher-level entities. For example, our SILK research system for prototyping UIs [Landay and Myers 1995] recognized a squiggle as text, a column of squiggles as a menu of text items, a row of squiggles as a menu bar, and squiggles next to circles or squares as radio buttons or checkboxes (see Figure 14.2). Similarly, the ASSIST system [Alvarado and Davis 2007] recognized circles as wheels and an arrow as the direction of gravity—to convert sketches into physics simulations (see Figure 14.3). Another example was shown in Figure 12.18, where Teddy recognizes gestures as three-dimensional shapes.

14.3.4 Gestures for Other Commands

Gestures continue to be widely used for navigation on devices. As discussed in Section 5.9, using gestures for scrolling dates back to at least 1991 with the Go flick-to-scroll gestures, which are widely used on today's devices as well. Sideways gestures on smartphones sometimes mean to go "back" or "forward," which has also been widely used. The Google Glass head-mounted display has a small rectangular touchpad on the right side (Figure 12.2) that recognizes gestures such as swiping forward and backward to navigate menus and other items, and swiping down to turn the device off. It also uses an accelerometer to detect head movements, so tilting the head up and down are gestures that are used to signal turning the device on and off. In virtual reality (VR) systems that recognize the pose of a user's hand, it is common for certain pointing poses or movements to represent flying or navigating in various directions.

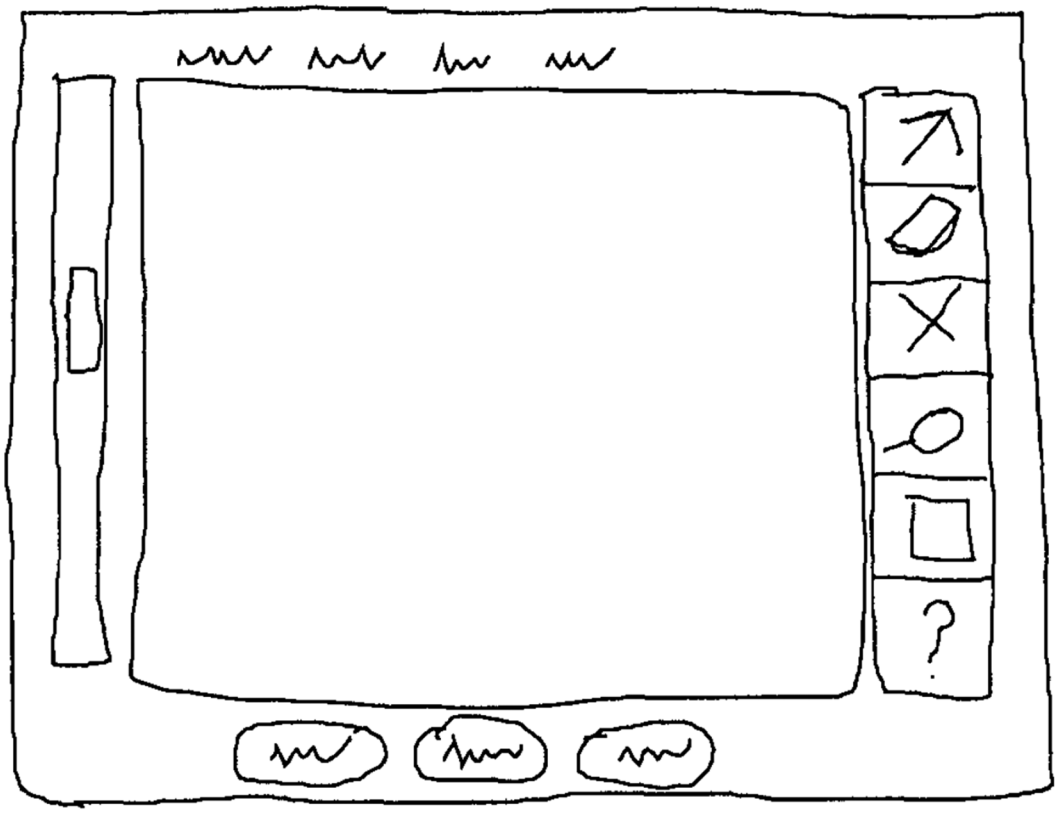


Figure 14.2 SILK recognized rectangles, circles, squiggles (as text), and other shapes. Combinations are recognized as other objects, such as two rectangles as a scroll bar (left), a row of squiggles as a menu bar (top), and squiggles in ovals as buttons (bottom). Source: Fig. 1 of Landay and Myers [1995].

14.4 Gesture Recognition Algorithms

A great deal of research and development has gone into the algorithms that recognize gestures. Especially with the hand printing and handwriting recognition, there has been tremendous commercial value in having a more accurate system, so there have been many patents issued in this area, on a wide variety of kinds of algorithms. Early systems used simple heuristics, but, like most other successful modern AI systems, current handwriting recognition algorithms use machine learning on a large corpus of examples.

For recognizing a collection of simple gestures, like swiping versus tapping, often an *ad hoc* recognizer will work. For example, the gestures may be easily differentiated by looking for specific *features*, like comparing the location of the first

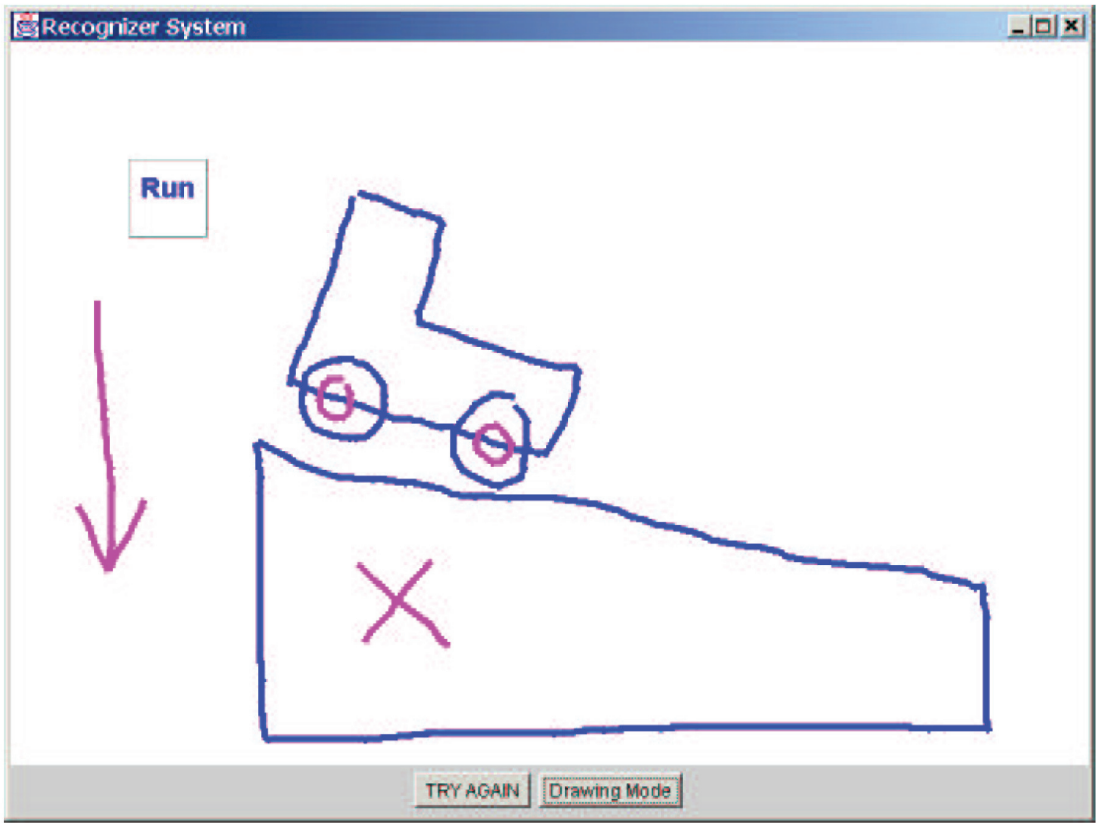


Figure 14.3 The ASSIST system recognizes sketches as physical items and runs a simulation. Source: Fig. 1 of Alvarado and Davis [2007].

and last points of the stroke and the difference in distance and time between them, which can be used to differentiate a tap from a long-press from swiping in the four main directions. This may be all that is needed for the primary built-in gestures on a smartphone. To add detection of, for example, dragging versus double-taps only complicates the algorithm a little.

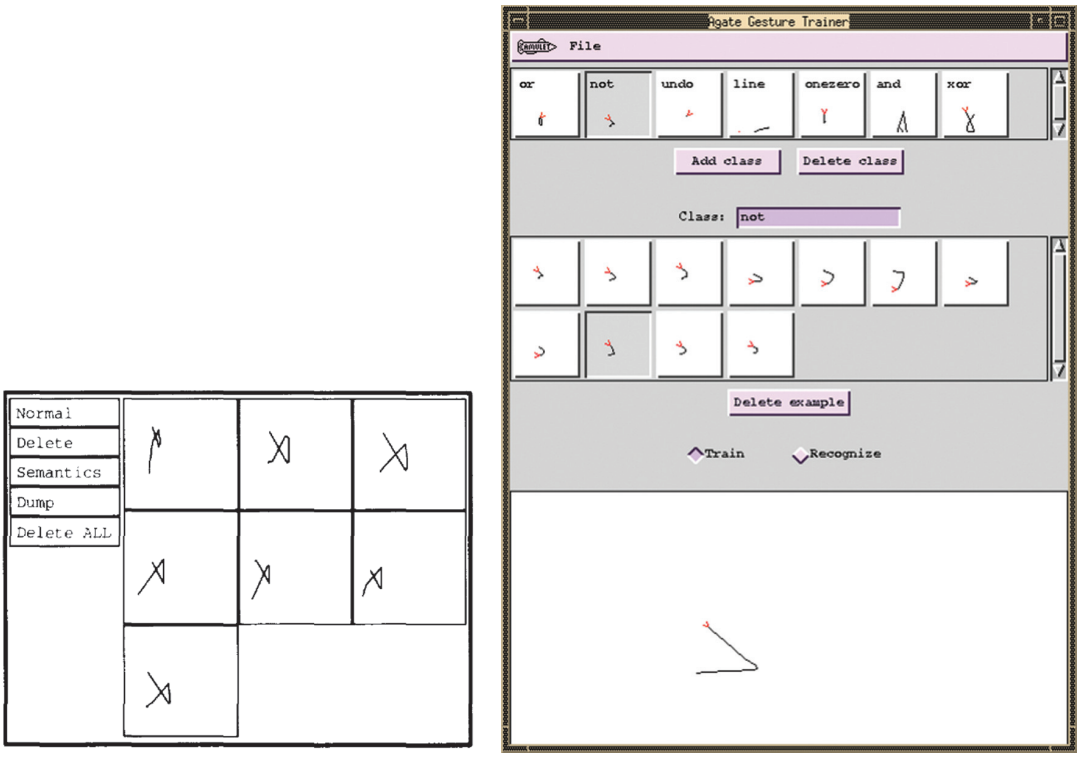
However, to support the recognition of different shapes, like a square versus a circle, requires detecting the sharpness of the angles at the corners, among possibly other features. While one can keep adding ad hoc code to differentiate each desired shape, this becomes more difficult as the gesture set becomes larger.

Dean Rubine developed a *trainable* gesture recognizer for single-stroke gestures as part of his Ph.D. dissertation in 1991 [Rubine 1991a, 1991c], and this algorithm was re-implemented and used in many later systems [Long et al. 2001], including as part of my Garnet and Amulet toolkits [Landay and Myers 1993, Myers et al. 1997a].

Rubine's algorithm used a set of 13 experimentally selected features that seemed to work well to differentiate the target gestures. These were:

> the cosine and the sine of the initial angle of the gesture, the length and the angle of the bounding box diagonal, the distance between the first and the last point, the cosine and the sine of the angle between the first and last point, the total gesture length, the total angle traversed, the sum of the absolute value of the angle at each mouse point, the sum of the squared value of those angles, the maximum speed (squared) of the gesture, and the duration of the gesture [Rubine 1991a].

The designer provided a small number, like 5 to 15, examples of the desired gesture (see Figure 14.4(a)), and the system computed the 13 values for each example, and then used machine learning statistical techniques to weigh the importance



(a) (b)

Figure 14.4 (a) Rubine's training tool with examples of a single-stroke "X" gesture. (b) The Agate tool from Amulet, which uses Rubine's algorithm, with a gesture set (top) and a new gesture being demonstrated (bottom). Sources: (a) Fig. 4 of Rubine [1991a]. (b) Screenshot by author.

of each feature to the set of the examples. This automatically took into account the desired variance in how the end-user would be able to draw the gesture. For example, if some of the examples were large and others were small, then size-based features would get a lower weight. Then, given a set of gestures and a target gesture to be recognized, the algorithm determined which was closest. There were also extensions for multi-stroke gestures, on-line recognition, and other variations.

An advantage of this algorithm was that the code did not have to change to define new gesture sets—the designer just gave a new set of examples. It was also easy to implement. This enabled us to create an interactive training tool we called Agate [Landay and Myers 1993] where the designer could give examples of the desired gestures, and the tool would create the data needed at run-time to integrate these gestures into the designer's application (Figure 14.4(b)). It was even possible to incorporate the Agate training tool into the running application, so the end-user could change the gestures.

Disadvantages of the Rubine algorithm included that it required floating point and fairly sophisticated math (including the Mahalanobis distance, discriminants, matrix inversions) to be calculated on the target gesture, which may be an issue for some platforms, and that the designer had only indirect control over what parameters were important. For instance, there was no way to directly say that the size did not matter or the gesture could be drawn clockwise or counterclockwise—the given examples must vary along these dimensions, which sometimes was not sufficient.

In 2007, Jake Wobbrock and colleagues published a new algorithm they called "A $1 Recognizer" that was much simpler but had higher accuracy on a core set of common gestures (Figure 14.5). This recognizer could be implemented using about 100 lines of code, and it would match the target gesture against a list of example gesture templates, where there only needed to be one example per kind of gesture. The key to the algorithm was that the target gesture and the examples were all normalized in four steps to remove variations:

> they are resampled, rotated once, scaled, and translated. Candidate points C are then scored against each set of template points T_i over a series of angular adjustments to C that finds its optimal angular alignment to T_i [Wobbrock et al. 2007].

Variants of this simple algorithm have been published that deal with one-dimensional gestures (lines or flicks), multi-stroke recognition [Vatavu et al. 2012], and so on.

There are additional complexities when supporting multi-finger gestures. For example, it is impossible for a person to put two or three fingers onto the touch surface at exactly the same time, so the system must avoid taking actions when seeing

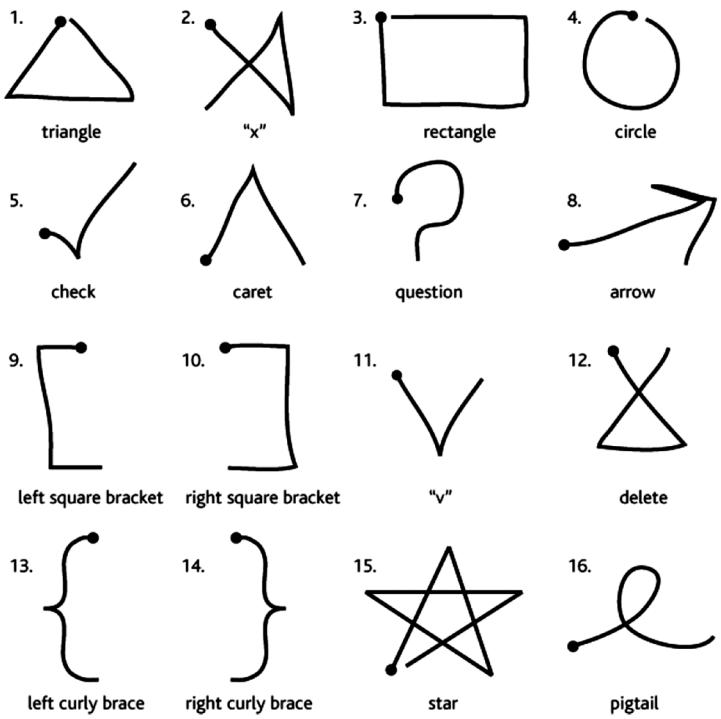


Figure 14.5 Gestures used to test the $1 recognizer. Source: Fig. 1 of Wobbrock et al. [2007].

the first touch(es) in case further touches happen within the appropriate waiting period. This is compounded for gestures such as three-finger-triple tap (which triggers the assistive zoom on an iPhone), or differentiating two finger swipe versus pinch, since these require even more timeouts and distance measuring. Research by Stephen Oney and me with others [Oney et al. 2019] provided an easier way to implement these using *touch groups* and *cross events*. The programmer could define a touch group with parameters such as the number of fingers, the maximum elapsed time between the first and last touches, and an area in which the touch must happen. Then, cross events defined movements of the finger(s) across relative or absolute lines or circles, optionally within a period of time. For example, a two-finger drag might be triggered when the fingers moved more than a specified distance relative to their touch points in any direction (so crossing a circle of that size). Or, a flick gesture might require crossing a line within a short time, which indicated that the finger was moving quickly. The research showed that a wide range of gesture behaviors can be represented succinctly and robustly with this architecture [Oney et al. 2019].

14.5 Evaluating and Improving the Learnability and Accuracy of Gestures

As will be discussed in Section 14.7, there are many considerations that the designer must take into account when designing a set of gestures. This section focuses on *learnability* and *accuracy*.

Learnability can be affected by how natural the gestures are. For example, Section 8.3.2 discussed how Graffiti was designed to be as similar as possible to the way that people naturally hand print characters, so the gestures would be familiar and easy to remember and learn, as opposed to Unistrokes where the gestures are selected purely based on frequency and speed, and hence appear arbitrary to users.

Jacob Wobbrock and I with others developed a technique [Wobbrock et al. 2005] to evaluate and improve the naturalness of the gestures as part of the design process for the EdgeWrite unistroke alphabet (Figure 8.9(b)). As a form of *participatory design*, we brought in users to the lab and explained the idea of the gestures to enter letters—by going from corner to corner of a square—and then asked the users to demonstrate how they thought each letter would be entered. In a second phase, we asked the users to iterate to provide new gestures when they accidentally used the same gesture for different letters (such as a vertical line for both "i" and "l"). In the end, we used all the gestures that users provided for each letter (so there are multiple ways to enter most letters), as long as they did not conflict with the gestures for other letters. This resulted in improving the initial guessability of the EdgeWrite gestures from 51.0% for our first design to 80.1% [Wobbrock et al. 2005].

Another key issue is the *accuracy* with which the system can recognize the given gestures. There are three error cases: the gesture not being recognized at all (called a *false negative*), which usually results in nothing happening or a quick error sound (beep); or the system thinking there is a gesture when the user did not intend to do a gesture at all (called a *false positive*); or else the gesture being *misrecognized* as a *different* gesture. In the latter case, a designer might want to modify the gesture set to make the two that were confused more distinct. However, even if the designer is familiar with the recognition algorithm, it may be difficult to identify how to improve the designs since what are similar with respect to the features may not correspond to how humans perceive the gestures. And designers who are not familiar with the algorithm may have little idea what would help. Research by Chris Long showed how a system called *Quill* might help [Long et al. 2001], in the context of the Rubine recognizer. Quill generated a *confusion matrix* showing how similar the recognizer thought each gesture was to all other gestures and provided guidance for what features were the culprits so designers could more easily determine what to change. In any case, the designer might want to consider the tradeoffs of

the three kinds of errors and adjust the thresholds in the algorithm based on the seriousness of the consequences of the errors [Lafreniere et al. 2021].

14.6 Funny

Back in 1991, a collection of researchers, including me, wrote a joke paper about "A nose gesture interface device: Extending virtual realities," which we presented at the annual ACM Symposium on User Interface Software and Technology (UIST'91) that year [Henry et al. 1991]. We talked about using one's nose to touch the screen (Figure 14.6) and to make gestures with one's nose, which was pretty silly with the CRTs available at the time, but has actually been cited 19 times, apparently at least a few times seriously, for new investigations of using one's nose to make gestures on a smartphone screen (e.g., Polacek et al. [2013]).

14.7 Summary of Considerations for Gesture Recognition

Many of the considerations for gestures have been discussed in the previous sections, including *naturalness*, *guessability*, *memorability*, *efficiency*, *accuracy*, *accessibility*, and *difficulty of implementation*. Many of these are affected by the *number of gestures* that are needed—a small gesture set (e.g., swipes in different directions) is much easier to design than a large set (e.g., all the letters in English).

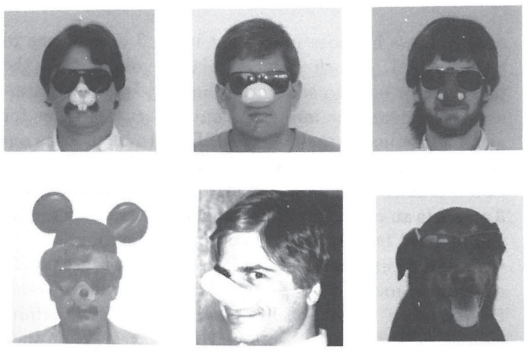


Figure 1: Alternative Noses: Bunny, Pig, Elephant, Mouse, Duck, Dog

(a)

(b)

Figure 14.6 (a) Shows the authors wearing toy noses—I am in the bottom left. (b) Rotating with one's nose on a CRT display. Sources: (a) Fig. 1 Henry et al. [1991]. (b) Color version of Fig. 3, reprinted with permission.

Since the design space of gestures is so large, Wobbrock, Morris, and Wilson provide a taxonomy of surface gestures [Wobbrock et al. 2009, p. 1086], which differentiates different features that the designer should consider (see Figure 14.7). For example, the *Form* of the gesture is whether the gesture is a static pose versus a dynamic path, as was discussed in Section 14.1. The *Nature* of the gesture describes how the user thinks about the gesture and how natural it is for users:

> [S]ymbolic gestures are visual depictions. Examples are tracing a caret ("^") to perform insert, or forming the O.K. pose on the table for accept. Physical gestures should ostensibly have the same effect on a table with physical objects. Metaphorical gestures occur when a gesture acts on, with, or like something else. Examples are tracing a finger in a circle to simulate a "scroll ring," using two fingers to "walk" across the screen, ... [or] swiping as if to turn a book page... Finally, abstract gestures have no symbolic, physical, or metaphorical connection to their referents. The mapping is arbitrary, which does not necessarily mean it is poor. Triple-tapping an object to delete it, for example, would be an abstract gesture. [Wobbrock et al. 2009, p. 1086]

Other considerations for the gesture designer were mentioned above in Section 14.4. The designer must decide if the *direction*, *speed*, and *size* of the gesture

TAXONOMY OF SURFACE GESTURES		
Form	*static pose*	Hand pose is held in one location.
	dynamic pose	Hand pose changes in one location.
	static pose and path	Hand pose is held as hand moves.
	dynamic pose and path	Hand pose changes as hand moves.
	one-point touch	Static pose with one finger.
	one-point path	Static pose & path with one finger.
Nature	*symbolic*	Gesture visually depicts a symbol.
	physical	Gesture acts physically on objects.
	metaphorical	Gesture indicates a metaphor.
	abstract	Gesture-referent mapping is arbitrary.
Binding	*object-centric*	Location defined w.r.t. object features.
	world-dependent	Location defined w.r.t. world features.
	world-independent	Location can ignore world features.
	mixed dependencies	World-independent plus another.
Flow	*discrete*	Response occurs *after* the user acts.
	continuous	Response occurs *while* the user acts.

Figure 14.7 Taxonomy of surface gestures. Source: Table 2 of Wobbrock et al. [2009].

matter. For example, the user might be allowed to draw an "i" as a single swipe up or down, but for scrolling, swipe up and swipe down should do different things. Similarly, drawing a rectangle slowly and swiftly might be the same, but a fast swipe gesture might cause faster scrolling than a slow swipe. And drawing a small circle or a large circle might both insert a circle in a drawing program but be differentiated as a lower- or uppercase "O" for a character recognizer.

14.8 Chapter Conclusions

Although handwriting and hand printing recognition is no longer a key use, gesture recognition continues to be important for applications. All mobile devices make extensive use of flicking in different directions and multi-finger gestures for scrolling and zooming, along with a small number of other easily distinguished gestures. Many drawing programs provide the option of neatening the results through gesture recognition of primitive shapes, and many VR/AR applications and some games identify poses and motions to control operations. Key usability challenges continue to be that the gestures are not discoverable and accurate recognition is a challenge, especially for people with disabilities. As more computing moves to tablets, VR/AR, and camera-based recognition, these issues will be increasingly important to address.

CHAPTER 15

Undo, Redo, Repeat, Cancel, Command History, and Macro Recording

15.1 Introduction

This chapter covers the interaction techniques related to *undoing* of previous operations and *cancelling* operations that are in progress. Normally, the undo operation is also related to *redoing* or *repeating* the operation, sometimes in a different context. Undo can be invoked on most desktops from a menu item in the "Edit" menu, or using Ctrl-Z or Command-Z, and redo is often Ctrl-Y or Ctrl-Shift-Z (or Command-Y or Command-Shift-Z). On an iPhone, undo can sometimes be performed by shaking the phone. Cancel is often available as the "Cancel" button in dialog boxes or by hitting the "Esc" keyboard key. Undo is rarely provided for web or smartphone operations, but sometimes the back button can serve as a form of undo. When there is a collection of previous operations, this can be used as a history of the interactions that the user has performed, which in some systems can be abstracted and reused as a macro.

Supporting cancel and undo are key ways to make a system more usable, in particular to make it easier for users to fix errors (such as when an object is deleted accidentally) and slips (such as incorrect typing). Providing undo also fosters exploration since users feel comfortable trying out operations if they know they can undo them if they do not like the results. Note that there is usually a central single way to perform undo (like Ctrl-Z) that undoes operations done in a plethora of ways (whether invoked through menus, direct manipulation, drag-and-drop, gestures, or whatever), so users can undo operations even if they don't even know exactly what operation they did (like drag-and-drop by accident into an unknown place). This is why providing undo and cancel have had prominent places in almost

all UI guidelines—for example, the original Macintosh Human Interface Guidelines from 1984 and Nielsen's 10 heuristics for evaluating the usability of interfaces [Nielsen and Molich 1990]. However, web and mobile devices have often omitted support for undo, making them more difficult to use than they should be [Norman and Tognazzini 2015] (see also Section 1.8.3).

15.2 Definitions

Undo is the operation that reverts a previous operation, to make it be as if that previous operation had not happened. The first time undo is executed, it undoes the previous operation, but subsequent "Undos" may do different things depending on the *undo model*, which describes what kind of undo is supported. If the system is using the *Single-Level Undo* model, then only the latest operation can be undone, so executing undo twice "Undoes the Undo" so that the original operation happens again. With *Multi-Level Undo*, executing undo multiple times undoes each of the previous operations in turn. The list of previously executed operations that can be undone is called the *undo stack* or *command history*.

In contrast, *cancel* causes the *current* operation to *abort* while it is in progress and should also restore the system to be as if the operation had not started. For example, most dialog boxes will have a "Cancel" button, which causes the operation that the dialog box describes to *not* happen. Similarly, when moving an object in a graphics editor, hitting the Esc key on the keyboard will often abort the move and put the object back where it was. This book uses the terms *abort* and *cancel* interchangeably, although most UI guidelines require the use of the word "Cancel" in the UI for consistency. More details about *Cancel* are covered in Section 15.4.

Redo has two completely different meanings. If an undo operation was just performed, then redo will "Undo the Undo" so the original operation that was undone is now in effect again. For example, if the user creates a square, and then hits Undo, the square will disappear. Hitting redo at that point would cause the square to reappear again, just where it was. In contrast, if the previous operation was *not* an undo, then redo becomes a *repeat* operation and performs the previous operation again. Performing redo or repeat right after creating the square will create *another* square, usually in a slightly different place so it can be seen. More details are in Section 15.7.

In *selective undo*, which is only supported by a few undo models, the user can select the specific operations in the past that are to be undone. Unlike other undo models, the selected operations do *not* need to be the most recent ones, so the user can skip and keep some operations while undoing previous ones. More information is in Section 15.8.3.

Some systems use the command history for more than undo. For example, a selection of these operations can be used as a *macro* or *script* that can be executed again as a group. Section 15.13 discusses some options.

15.3 History of Undo/Redo

Although writing instruments date back to pre-history, the idea of putting an eraser on the end of a pencil dates to the late 1850s, possibly the first example of integrating undo [Lipman 1858]. To "Undo" ink from a pen or typewriter, Bette Nesmith Graham in the 1950s invented "Liquid Paper," a form of quick-drying white paint that can be painted over errors (she is also notable as the mother of Michael Nesmith, who became famous as part of the 1960s rock band and TV show "The Monkees"). Typewriters provided a ribbon with ink and a separate ribbon with white that could be used to fill in incorrect letters, starting in about the 1970s.

The idea of providing an undo operation in computers appeared in the influential MIT Ph.D. dissertation of Warren Teitelman on *Pilot* in 1966, where the future work describes that when "a transformation was applied that should not have been ... we should have it back up and 'undo' some of the transformations it executed" [Teitelman 1966, p. 145]. However, Andy van Dam says that his FRESS hypertext system from 1968 was the first to actually *implement* undo, and it even supported the multi-level model:

> The most popular feature, however, was undo. I will claim that, to the best of my knowledge, FRESS was the first system to have an undo. We saved every edit in a shadow version of the data structure, and that allowed us to do both an autosave and an undo. I think the most important feature in any system built today has to be indefinite undo and redo. One level is better than zero, but not enough, and wholly inadequate for serious work. It doesn't matter how it's done, whether it's journaling, transcripting done with inverses, whatever, but you've got to have it. [van Dam 1988]

Warren Teitelman went to work at the BBN company after getting his Ph.D. from MIT, and while he was there created a version of the LISP programming language called BBN-LISP. He incorporated the undo feature into the editor for BBN-LISP in 1970 [Teitelman 2008]. Teitelman became one of the founding members of Xerox PARC in 1972, where he created Interlisp, which also supported undo.

The Bravo WYSIWYG text editor and formatter, created in 1974 (see Section 2.4.3.2), supported a single level of undo, which was surprisingly limited. It only worked if one had not moved the selection, and there were a variety of commands that could not be undone. Since it was a single level, executing the undo again would *redo* (undo-the-undo). The keyboard Esc key was actually used to *repeat*

the previous operation with the same parameters, but at the new cursor point—for example, to "insert 'abc'" multiple times in different places [Xerox 1978]. In 1977, researchers from IBM wrote an influential article in favor of undo:

> It would be quite useful to permit users to 'take back' at least the immediately preceding command (by issuing some special 'undo' command). ... [T]he benefit to the user in having—even just knowing of—the capability to withdraw a command could be quite important (e.g., easing the acute distress often experienced by new users, who are worried about 'doing something wrong'). [Miller and Thomas 1977, p. 522]

Meanwhile, as discussed in Section 11.3, the Smalltalk group at Xerox PARC was inventing covered windows in their environment, which also supported Cancel and Undo in the standard popup edit menu at least by 1976 (see the menu in Figure 2.10(b)). Undo was used throughout the many PARC systems. For example, when Teitelman saw the Smalltalk implementation of multiple windows in 1976, he was inspired to create a window manager for Interlisp, which he demonstrated in 1977, and called the "Display oriented programmer's assistant," which also provided Undo [Teitelman 1979, 2008].

The Xerox Star from 1981 (see Section 2.4.4) used a special keyboard, with a large "UNDO" key on the right hand side (see Figure 15.1). There was also a "STOP" key there, as well as a "CANCEL" button on some dialog boxes (see Figures 7.8 and 9.2). I haven't found any definitive discussion, but the information I did find suggests that it was only a single-level undo:

> UNDO reverses the effects of the last command. It provides protection against mistakes, making the system more forgiving and user-friendly. Only a few commands cannot be repeated or undone. [Smith et al. 1982a, p. 527]

With the Apple Lisa and Macintosh (Section 2.4.7), the Undo command became generally required, with a prominent place at the top of the Edit menu (see Figure 5.11). Originally, the Lisa and Macintosh only supported a single-level undo, and the "Redo" command replaced the "Undo" command at the top of the menu after an undo was performed. Later, the Macintosh switched to multi-level linear undo, and then both Undo and Redo commands started to be on the top of the Edit menu.

Most Lisa and Macintosh menu items have accelerator keys, which are shown in the menu to remind users of what they are. There is some controversy around how the accelerator for Undo became "Control-Z". A *New York Times* article from 2009 (and quoted in the Wikipedia article on Undo as of August 2020) says: "The undo command ... [was] assigned the now-familiar keyboard shortcut of Control-Z

Figure 15.1 Special keys on the right side of the Xerox Star keyboard, including the "UNDO" key. See also Figure 9.1 for the rest of the keyboard. Source: Courtesy DigiBarn Computer Museum, http://www.digibarn.com/friends/curbow/star/keyboard/index.html.

by programmers at the research center Xerox PARC" [Zimmer 2009]. This is incorrect. The Control-Z shortcut was instead selected by Larry Tesler of Apple for the Lisa in 1983. Larry Tesler explained to me in an email from March 30, 2014 that (see also Section 28.2):

> The Lisa was the first system to assign XCVZ to cut, copy, paste and undo (shifted with the "apple" key). I chose them myself. X was a standard symbol of deletion. C was the first letter of Copy. V was an upside down caret and apparently meant Insert in at least one earlier editor. Z was next to X, C, and V on the U.S. QWERTY keyboard. But its shape also symbolized the "Do-Undo-Redo" triad: top rightward stroke = step forward; middle leftward stroke = step back; bottom rightward stroke = step forward again. So X, C, and even V were not novel. But I know of no use of Z to mean Undo before the Lisa. When Z undid only the most recent command, another Z redid it.

As discussed above (Section 2.4.8), Microsoft Windows used the same accelerator keys as the Macintosh and Lisa. Multi-level undo came to the Macintosh and Windows in the late 1980s, with the divergence of the keys for Redo—Macintosh using Command-Shift-Z and Windows using Ctrl-Y.

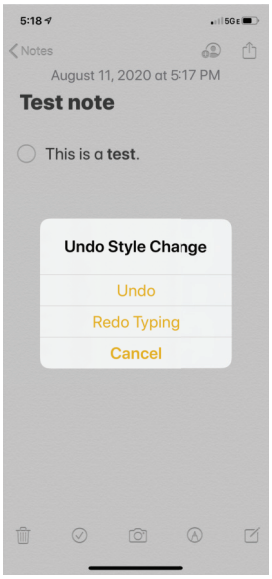


Figure 15.2 iPhone iOS 13.6 multi-level undo in Notes: I did style-change, then text edit, then undo (of the text edit), and when I shake the device again, it offers to redo the text edit (undo the undo) or else to undo the previous style change. Source: iOS screen capture.

The pen computers and PDAs of the 1990s generally also supported undo in the same way. The Palm provided Undo at the top of the Edit menu, but its accelerator key was "U" (see Figure 7.7(a)). The iPhone originally did not support undo at all. With the iOS 3.0 update in 2009, it added the ability to shake the phone to cause undo, sensed by the iPhone's accelerometer. This was a single-level undo/redo originally, later changing to support multi-level undo (see Figure 15.2). Android has a universal "back" button on the bottom control panel, but it serves more as a "Cancel" than undo (see Section 15.9). Similarly, web applications rarely support undo directly, but sometimes the back button can be used to perform an analogous operation (Section 15.9). In fact, Don Norman [Norman and Tognazzini 2015] and Alan Kay[1] (among many others) have bemoaned the fact that most of the apps that people use today do not support undo anymore.

15.4 Cancel or Abort

UI guidelines have always called for an ability to *cancel* (*abort*) long-running activities, including anything that required a progress bar (Chapter 6, see, e.g.,

1. Alan Kay said this in a panel on May 12, 2016, at the ACM CHI'2016 conference, which I tweeted about: https://twitter.com/bradamyers/status/730794058411474945.

Figure 6.4). In fact, the cancel operation has been discussed in many previous chapters since cancelling is relevant to almost all interaction techniques:

- Section 1.8.3 argues that the need for Cancel and Undo is one of the reasons that UIs are hard to design.
- Section 5.2 covers cancelling an auto-scroll by hitting the Esc key, which is actually more of a *stop* since it does not go back to where the document was before the scrolling started, like a real cancel would.
- Section 7.3.5 covers aborting menu selections.
- Section 7.3.8 covers a Cancel button, usually in a dialog box, and how it might be the default for dangerous operations (instead of OK), for example, for reformatting a disk. The accelerator key for the Cancel button is the Esc keyboard key.
- Section 8.4 covers cancelling auto-fill or autocomplete in text entry.
- Chapter 9 about dialog boxes discusses Cancel in many places. For example, Sections 9.2 and 9.3 discuss Cancel in dialog boxes, and Section 9.10 discusses how much is canceled.
- Section 10.6.2 discusses aborting graphical operations, like creating or resizing a rectangle, while in progress.

In addition, there are global abort commands in many systems. For example, Ctrl-C or ^C ("C" for "Cancel") was used to abort the current operation in command line interfaces (Section 2.5.1.1), especially on the old Digital Equipment Corporation (DEC) TOPS-10 and TOPS-20 operating systems (from the late 1960s and 1970s), and then in all Unix systems (from the 1970s through today), and also in Microsoft DOS (starting in 1981, and continuing in Windows as the "Command" or "cmd.exe" window). Since Apple's merger with NeXT and adoption of NeXT's Unix-based operating system in 1999 (see 2.4.11), the Macintosh has had a Terminal window that also uses ^C for abort while still using Command-C for Copy (on the Macintosh keyboard, Command and Ctrl are separate keys).

On IBM PC compatible computers, starting from around 1981, the accelerator key Ctrl-Alt-Delete (Ctrl-Alt-Del) interrupts the running program. Originally, it just rebooted the computer, but starting from Windows 95, it provided a variety of options. Today, for example, after hitting Ctrl-Alt-Del, the user can choose to reboot the computer, abort the currently running program, bring up the task manager to review running applications, and so on.

On the Macintosh, the traditional abort key was Command-Period (Cmd-dot), which primarily was used for long-running modal tasks such as file transfer or

printing. It rarely worked to abort programs that were stuck. Command-Option-Esc brings up a Force Quit dialog box for terminating processes. In Classic Mac OS, it let you terminate the frontmost process (and did not let you pick which one), but it is now more flexible, like the modern Ctrl-Alt-Del on PCs.

On mobile devices, the iPhone has always had the one "home" button (or home gesture), which always exits the current application and returns to the home screen, thereby serving as a universal cancel (see Figure 3.26). Similarly, Android provides the "home" button at the bottom. As a "super-abort," most computers and consumer electronics devices will turn off ("shutdown") if one holds down the power button for more than 10 or 15 seconds.

15.5 Unit of Undo

When the user hits the Undo key, what is undone? The obvious answer is the *one* last operation, but this is not always the most usable design, and often a single undo will undo *multiple* operations. For example, when the user is typing in a text editor, and hits Undo, should the previous letter be undone, or the previous word, or even a larger unit? Undoing by characters is the most flexible, but it is *not* what most editors do since it is too tedious for a user, and if the user wanted just the previous character undone, then the backspace key could be used. Therefore, modern text editors have a complex algorithm for determining how many characters to undo, which takes into account pauses in typing, explicit movements of the cursor, use of the backspace key, the characters being typed (e.g., whether an Enter key has been typed), and so on. When typing multiple backspaces, they would typically be all undone in a single operation, to restore all of the deleted text. Another example is find-and-replace, which when replacing multiple instances are undone all together in some editors, whereas other editors undo each replace individually.

Similarly, in a graphics editor, there are often multiple commands aggregated for Undo. For example, in PowerPoint and other editors, one can move the selected object using the arrow keys by a small increment for each press of the key. Holding down the arrow key causes auto-repeat, so the object continues to move. This is called *nudging* the graphics (see Section 10.6.3). PowerPoint undoes the entire set of moves in one direction with a single undo operation (if the user switches to a different arrow key and moves in a different direction, that requires its own separate undo operation).

In the opposite dimension, a single keystroke sometimes causes multiple operations, which then can be separately undone. The classic example is autocorrect in Microsoft Word and now other editors—if the user types "aaa. teh<space>" then Word will autocorrect the "teh" to be "the", automatically capitalize it to be "The",

and finally insert the space, to end up with “aaa. The<space>”, all triggered by typing the one <space> character. After typing the <space>, hitting Undo one time will first undo the capitalization, resulting in “aaa. the<space>”; hitting Undo again undoes the autocorrect: “aaa. teh<space>”, and finally a third time undoes the <space>. So even though the user entered a single editing operation (hitting the space key), multiple undo operations are needed to undo everything it causes.

15.6 What Can Be Undone

A related question is which operations can be undone at all? One might imagine that the best option would be if the user could undo all operations, but this is clearly not possible and sometimes may not even be the most useable.

Typically, there will be operations that *cannot* be undone such as sending an email, printing a document, saving a file, or reformatting a disk. In older systems, when such an operation was on the undo stack, then when the user had undone back to this point, no further undoing was possible, which seems to make sense. However, it is actually useful for the user to be able to continue to undo in certain cases. Imagine editing a text file, and saving occasionally for safety. If the user wants to do a sequence of edits, a file save might be in the middle of that sequence. Therefore, Microsoft Word and other editors now do *not* add the Save to the undo stack at all, so the Undo command can affect the edits that happened *before* the save, even though the save itself cannot be undone (so basically the save operation is just skipped in the undo sequence). It is also important to note that the undo stack is rarely, if ever, saved with a file, so the undo stack is lost if the user closes the edit window for a file.

There are other operations that designers have decided are better *not* to have in the undo stack, even though they could be. In particular, scrolling and selection operations are almost never undoable, and neither is moving around or resizing of windows. If you want to *scroll* somewhere in a document, and then go back to where you were, sometimes the scrolling can be canceled if you had not released a drag operation. If the scroll happened due to navigation (like a hyperlink within the document), then often *command-leftarrow* or *control-leftarrow* will go back, but this is a separate operation from Undo, and this rarely works for navigating using the scrollbar. Interestingly, in a study my group did of coding tasks, programmers spent 35% of their entire time in redundant scrolling and other navigations through the document, often trying to get back to where they were [Ko et al. 2006]. I personally have written a “bookmark” macro command for the Emacs and Microsoft Word text editors that allows me to go back with a single keystroke to a location that I marked with a different keystroke, which I find invaluable. Thus, an undo of scrolling might be useful in many situations, even though no commercial system supports it.

One of the fundamental principles of text and graphical editors is that *selections* are ephemeral and easy to do, so it makes sense that they might not be put onto the undo stack. However, sometimes it is tricky to get the desired objects selected, and a slip might accidentally mess up the laboriously created selection, so an undo might be useful in certain cases. In a research system my group created, we discovered that allowing selections to be undone enabled more useful scripts in a graphics editor (see Section 15.13). A related issue is that programs are not necessarily consistent or sufficiently careful to restore the selections when undo operations happen, which can be frustrating—for example, in the Microsoft Outlook email program (version 2016), if one deletes an email message, changes the selection, and then undoes the delete, the restored email message is *not* selected (the selection doesn't change) making it difficult to find the restored message (most other programs select the un-deleted object).

The clipboard's *cut-copy-paste* operations have an interesting interaction with Undo. If one undoes a cut operation, obviously the object is restored onto the screen, *but the clipboard is not restored*. That is, consider the following:

1. Select circle
2. Copy
3. Select square
4. Cut
5. Undo

Operation 5 will undo the "Cut" in that the square will reappear, but the clipboard still holds a square, so a paste operation will paste a square (and not a circle). In fact, the "Copy" operation is not even entered into the undo stack. There are various reasons for this, the primary one being that the clipboard is global across all applications, whereas Undo is local to a single application. There are also some clever interactions that take advantage of this behavior, such as to move content that was entered in the middle of a sequence of operations that all need to be undone.

15.7 Redo/Repeat

The user might undo an operation by mistake, so there needs to be a way for the user to fix that mistake too, which is why almost all systems provide a *redo* that *undoes-the-undo*, restoring the system back to the way it was.

If the previous operation was *not* an undo, then *redo* may instead do a *repeat* operation, which redoes the previous operation again. Often the repeat operation

will operate *on the currently selected object*, which may be different than the originally selected object (since changing the selection does not go into the command history). This is very useful for when the user wants to repeat an operation many times. For example, to apply a particular font to many parts of a text document, the user can select the first part and select the font, then select each part in turn and use the Repeat operation. If the operation being repeated is a Create, then most systems will slightly offset the new objects so that they are in a visibly different place and the user can see that something happened.

Both Redo and Repeat use the same accelerator key (usually Ctrl-Y or Ctrl-Shift-Z), and sometimes use the same label (usually "Redo"), but other times, the label of the operation changes between "Redo" and "Repeat" (or an icon changes in Microsoft Office 2016 from to) depending on whether the previous operation was an undo.

15.8 Undo/Redo Models

There are three basic models for Undo, which have sub-options. The simplest is single-level undo, where only the previous operation is remembered. Next is multi-level undo, with a list of operations. Finally, selective undo, where there is a list, but the user can select various operations, which are not necessarily the previous ones, to undo.

15.8.1 Single-level Undo

The single-level undo model is fairly simple—if the previous operation is undoable, then undo is provided, otherwise it is not. Hitting an Undo a second time undoes the undo, so the single operation is used for both undo and redo. For example, repeatedly hitting ^Z would toggle the previous operation as done and not done. The advantage of single-level undo is its simplicity and also the low memory costs since only the previous operation needs to be remembered.

Although almost all applications moved to multi-level undo when there became sufficient memory, some *still* only support single-level undo. For example, Microsoft Outlook email manager (version 2016) only provides for single-level undo, and many operations are not undoable. Google's Gmail browser app currently (2022) displays a floating button for Undo at the lower left of the window for the last action, but it disappears after a short amount of time. Interestingly, Adobe Photoshop supports *both* single- and multi-level undo. For a long time, Ctrl-Z toggled the previous command (single-level undo), and Ctrl-Shift-Z went forward through the undo stack, while Ctrl-Alt-Z went backward (multi-level undo). This is because artists find it useful to be able to quickly flick back and forth to the previous image edit since this helps the changes stand out. Recently, Photoshop

changed the default key assignments to be more standard: Ctrl-Z for regular linear undo, Ctrl-Shift-Z for Redo, and Alt-Ctrl-Z for "Toggle Last State".

15.8.2 Multi-level Undo

As discussed above, multi-level undo has been supported since at least 1968 with FRESS and was available in various tools like Emacs since the 1970s. But many systems found it too expensive especially in memory, so it was temporarily not available in the 1980s even though everyone knew it was desirable. By the 1990s, though, multi-level undo was ubiquitous, almost always using the linear model, but other models have been investigated, including the Emacs model and the tree model described below. Other multi-level undo models have been explored by researchers [Vitter 1984, Yang 1988] but proved to be too complex for users [Berlage 1994]. The next few sections discuss these undo models.

15.8.2.1 Linear Model

It seems clear that when multiple undo is supported, executing undo multiple times in a row should undo the previous operations, from most recent backward to next most recent, and all the way back to the oldest (least recent). And all of the multi-level undo models do this. The key distinction comes about when the user stops undoing, and then does something new.

For example, in Figure 15.3, the user performs four operations: A: create a green rectangle, B: resize that rectangle, C: make the rectangle be blue, and then D: rotate the rectangle; the result is shown at the right of row 1. At this point, the undo stack has four items in it. Then the user performs an Undo, which removes the effect of D, so the rectangle is no longer rotated, as shown at the end of row 2. The undo stack saves operation D, however, in case the user later does a redo, so the undo stack continues to contain the four operations. Suppose the user then calls undo a second time, so the change-to-blue operation C is undone, so the rectangle goes back to being green, as shown in row 3. Now, suppose the user does a Redo, which restores operation C, so the rectangle is once again blue, as shown in row 4. If the user did another redo at this point, then the rectangle would rotate (operation D), or if the user did an undo at this point, then that would undo the change color again, removing C, as in row 3.

But instead, suppose the user now does a *new* operation, like an add shadow, as shown in row 5. In the linear model, the operation D would be *discarded* off of the undo stack, and replaced with operation E, so the undo stack always only contains linear list of operations. If the user had done a new operation after row 3, then both C and D would have been discarded. This also means that there are states that

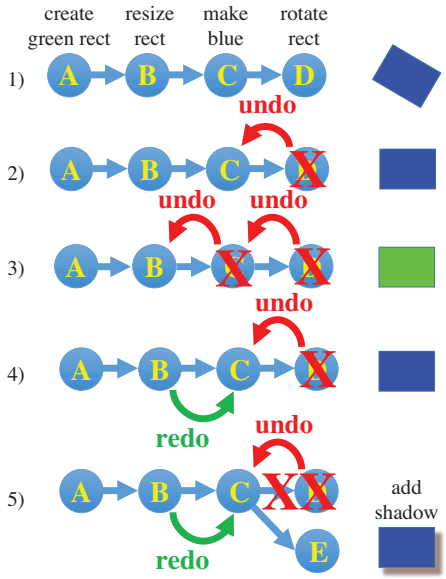


Figure 15.3 Example of linear undo. Source: Original by author.

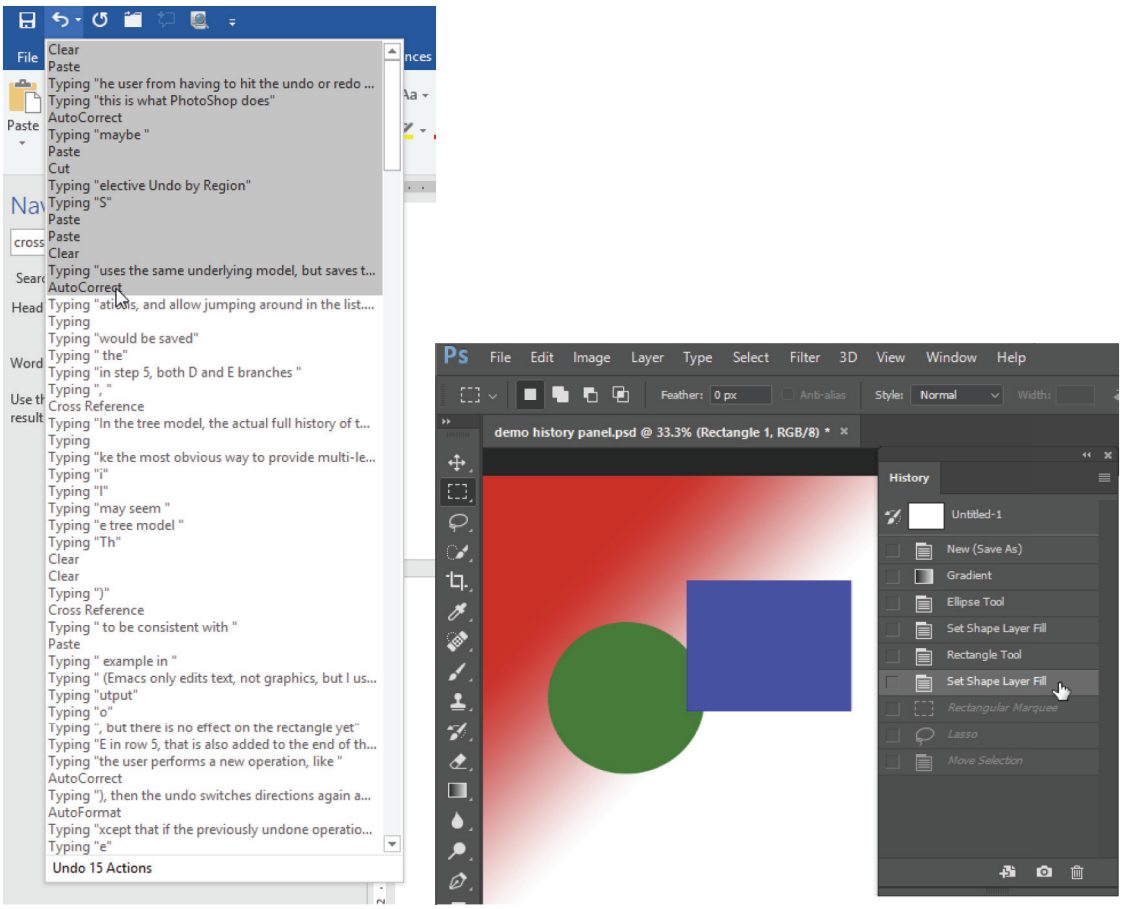
the document was in that the user can no longer get back to (except by manually remembering and repeating the operations)—here the state after row 3.

There is no way with the linear model to get the effect of operation D but remove the effect of operation C (one would need selective undo for that, see Section 15.8.3).

A slight extension of the linear model is to show the user the full list of operations and allow jumping around in the list. This uses the same underlying model but saves the user from having to hit the Undo or Redo command a bunch of times. For example, Microsoft Office 2016 has a popup menu of the previous commands (newest at the top), which the user can move down through to undo many operations at once (Figure 15.4(a)), and Photoshop has the History Panel, with the newest operation at the *bottom*, which supports similar operations (Figure 15.4(b)).

15.8.2.2 Emacs Model

The undo model used by the Emacs text editor [Stallman 1979] from around 1974 supports multi-level undo in a different way that has not been used by any other significant system that I am aware of. The key advantage of the Emacs model is that all previous states that the user saw can be returned to (unlike the linear model). The disadvantage is that it is a more complex model that the user must understand.



(a) (b)

Figure 15.4 (a) Office 2016 undo stack. (b) Adobe® Photoshop® 2017 History Panel (far right). Sources: (a) Office Word 2016, used with permission from Microsoft. (b) Adobe product screenshot reprinted with permission from Adobe.

In the Emacs undo model, there is a marker in the undo stack, shown by the black vertical arrow in Figure 15.5 (Emacs only edits text, not graphics, but I used the rectangle example in Figure 15.5 to be consistent with Figure 15.3). Before doing any undos, the marker is always after the most recent operation, as shown in row 1. When the user does an undo, the marker keeps track of where in the undo stack the user is. To perform the undo, the *inverse* of the operation is appended to the end of the undo stack. So after one undo, an un-rotate rectangle operation (indicated by D') is added to the end of the undo stack in row 2, and the marker moves to the left by one operation. Similarly, after another undo, the rectangle is

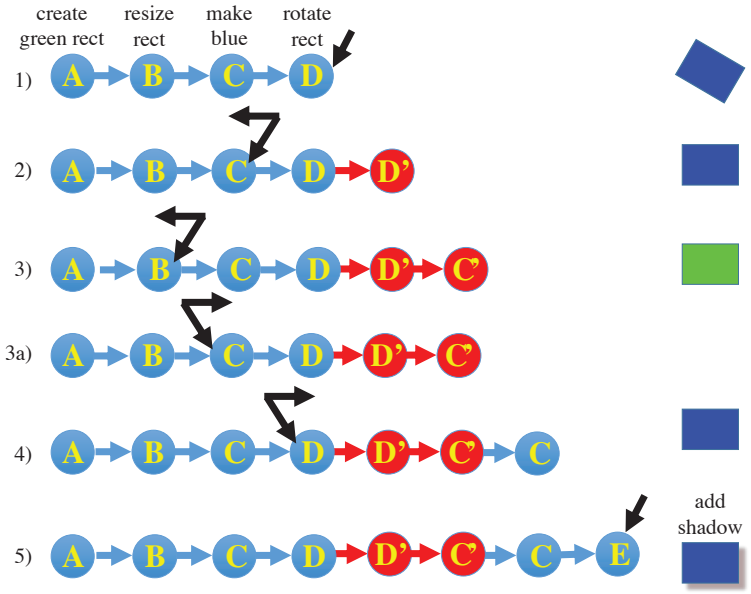


Figure 15.5 Example of Emacs undo. Source: Original by author.

returned to green, and marker moves one more. In Emacs, rather than have two different commands for Undo and Redo, the abort command (usually ^G in Emacs, but different versions of Emacs have different key assignments, and the user can always override these anyway) reverses the direction of the one Undo command (usually ^Z). So in row 3a, the user has done the abort command (^G), so now the marker is ready to go to the right, redoing the operations, but there is no effect on the output yet. Again, the redone operations are appended to the end of the history, so in row 4 the make blue operation (C) is added to the end of the history. Continuing to invoke ^Z would move through the operations to the right, except that if the previously undone operations are hit (here D'), then the undo switches directions again and goes backward, as if the user had hit ^G. When the user performs a new operation, like E in row 5, that is also added to the end of the history, but the marker is then moved to the end as well. Now, if the user starts doing undo again, then the marker moves backward and can access all the previously seen states.

15.8.2.3 Tree Model

In the tree model, the actual full history of the commands is saved. For example, in Figure 15.3, in step 5, both the D and E branches would be saved, and when doing redo from C (moving to the right), the user must pick which branch to use. This may seem like the most obvious way to provide multi-level undo since it preserves all

past history and also matches the actual model of what the user did. However, it has proven very difficult to use primarily because there are so many small operations, the tree gets to be enormous, and the labels on the branches are not very helpful. For example, Figure 15.6 shows the tree visualization provided by the undo-tree.el package [Cubitt 2020] for just a few operations.

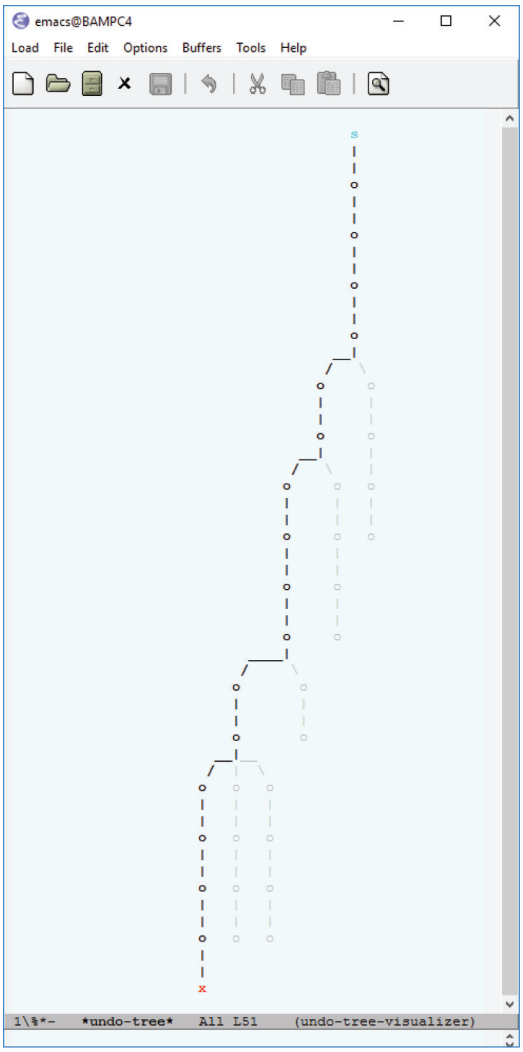


Figure 15.6 Example of Emac editor displaying the Tree Undo visualization. The dark path on the left is what is in effect now, and the lighter paths are available. Source: Screen capture of GNU Emacs 24.3.1 (i386-mingw-nt6.2.9200) of March 17, 2013, on MARVIN, running undo-tree.el Version: 0.7.4 [Cubitt 2020].

15.8.3 Selective Undo

The idea of selective undo is appealing and seems useful, but it has difficulties in practice. The basic idea is that the user should be able to *select* a particular operation or operations in the past, and undo *only them*. Equivalently, the user might pick specific operations in the undo stack to skip over or leave intact, while then undoing earlier operations. The problem with selective undo is that the *meaning* of undo is not clear. For example, consider Figure 15.7—it seems clear what should happen when the "make blue" operation (C) is selectively undone—the object does not turn blue (row 2a), but what happens if the create rectangle operation (A) is undone (row 2b)? Then operations B, C, and D have nothing to operate on. Another problem is illustrated in row 1x, where the last operation is another color change. Then undoing operation C has no visible effect since it is hidden by operation D. Different selective undo models make different choices about how to deal with these and other issues.

Another issue with selective undo is that the user must be given some way to *select* which operation to undo. This is usually provided by displaying a list of previous operations (as in Figures 15.4, 15.6, 15.8, 15.9, or 15.10) and allowing the user to select which one or ones to undo. The problem is that there may be hundreds of similar looking operations in these lists (see especially Figure 15.4), so users must be given some way to differentiate which is desired.

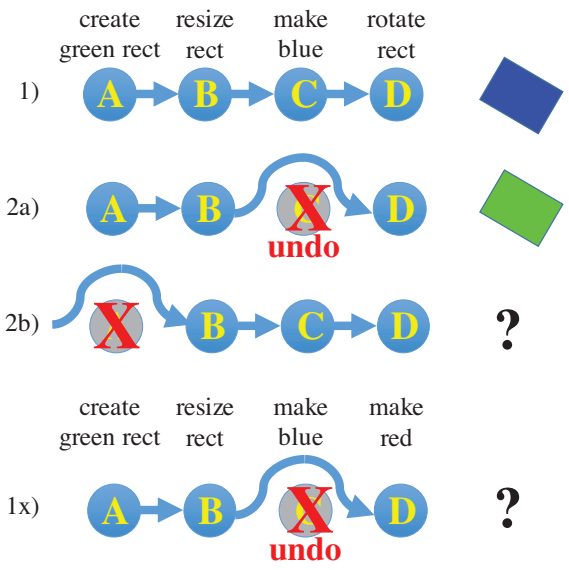


Figure 15.7 Selective undo. Source: Original by author.

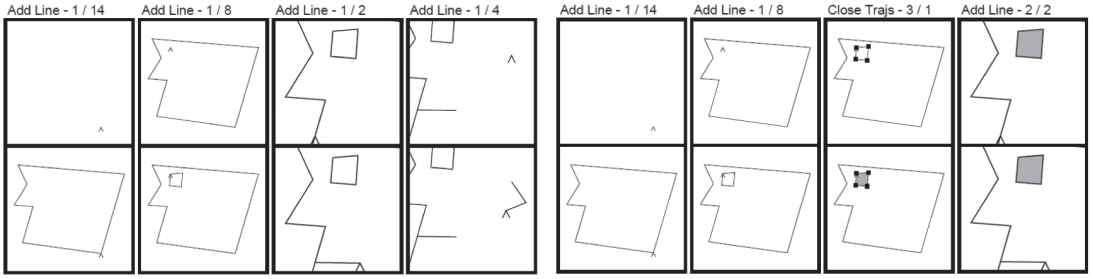


Figure 15.8 Chimera script model for undo/redo—adding a new "Close Trajs" operation in the past. In the graphical History Panel, each vertical pair shows the "before" and "after" of each operation. (a) The original history, and (b) after editing the past to make the eye gray. Subsequent operations, like drawing the mouth, are redone. Source: Excerpts of Figs. 3 and 5 of Kurlander and Feiner [1988].

15.8.3.1 Selective Undo by Region

Selective undo by region is the simplest way to support selective undo. This supports regular linear undo in a user-selected area of the document and thus avoids the questions of Figure 15.7 since only linear undo is supported for the specific rectangle. However, if there were operations on a *different* rectangle interspersed with the operations on this rectangle, then each of these rectangles could have their operations separately undone. The only complexity of this approach is what to do about objects that cross the boundary of the selected region—do they get all, partially, or not undone?

Emacs supports this kind of selective undo, where the user can undo in a user-selected region of the document. Operations that span across the selected and non-selected regions cause the selective undo to abort (so selective undo is not available).

Photoshop provides the History Brush that provides selective undo by region. The user opens the History Panel (Figure 15.4(b)) and selects the set of operations to be undone with the History Brush. Then, wherever the user paints over the image, that specific area will be returned to the way that it was in the selected state in the History Panel. Note that since this operates at the pixel level, and pixels are always entirely in or out of the History Brush, the problem of selectively applying an operation to a part of an item cannot happen. However, our interviews of a few professional and student designers suggested that the Photoshop History Brush and History Panel are difficult to use and often don't do what the designers wanted [Myers et al. 2015]. Another way that Photoshop supports selective undo is through its layer model—the user can limit the undo to only operate on the selected

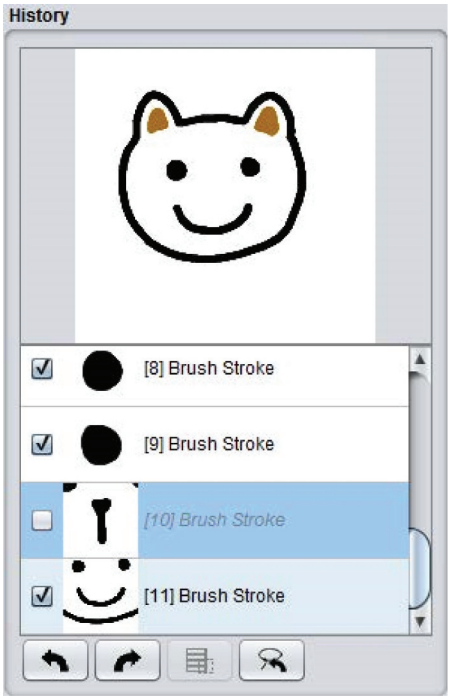


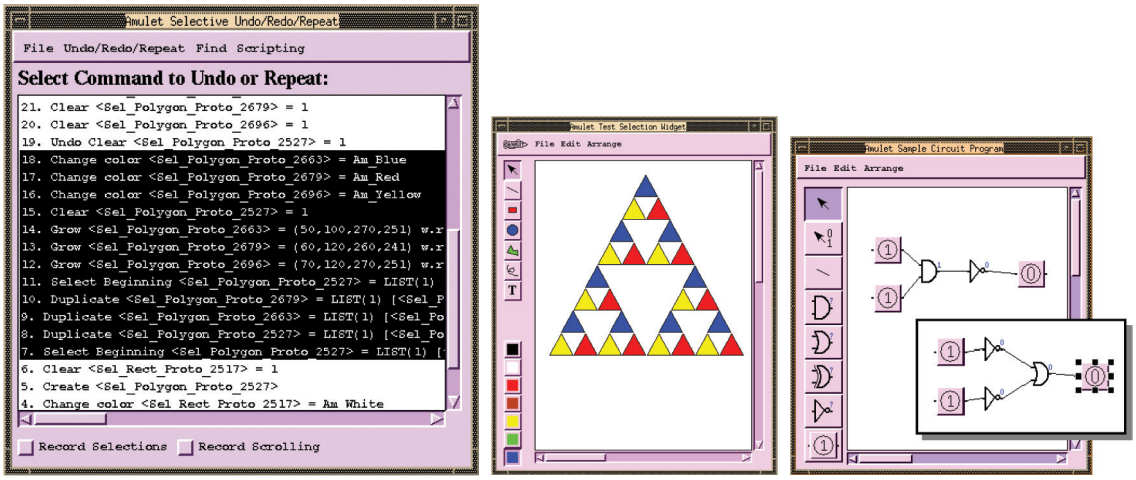
Figure 15.9 Aquamarine's History Panel showing operation #10 (brush stroke for the nose) selectively undone. Source: Fig. 1 of Myers et al. [2015].

layers, ignoring operations on other layers. Since the layers are quite independent of each other, this avoids the complications of interactions among the operations that are undone and those that are not undone.

We investigated a different form of regional selective undo for text in the Azurite plugin [Yoon and Myers 2015] for the Eclipse code editor. Here, if the user wanted to undo a region of code, a timeline view of all the code edits could be used to scrub back and forth to see what the code looked like at a certain point in time in the past, in a "diff" view (Figure 15.11). Azurite handled situations where edits spanned the selected and not-selected regions, with the fallback of asking the user what to do.

15.8.3.2 Script Model

The *script model* for selective undo requires that the effect of an undo should be the same as if the undone action had never occurred in the first place [Berlage 1994]. Although this sounds simple, it has the problems and ambiguities discussed above and shown in Figure 15.7. An advantage of the script model is that it may also support the *addition* of new operations in the past (not just undoing previously performed operation). For example, one might insert a change-border



(a) (b) (c)

Figure 15.10 (a) In Topaz, the user can select operations (which do not have to be contiguous). to be part of a macro. (b) Executing the Sierpiñski Gasket macro 10 times. (c) A macro that converts an And gate and a Not gate into two Not gates and an Or gate and reconnects the wires (De Morgan's law). Source: Color versions of Figs. 5, 1, and 2 of Myers [1998b].

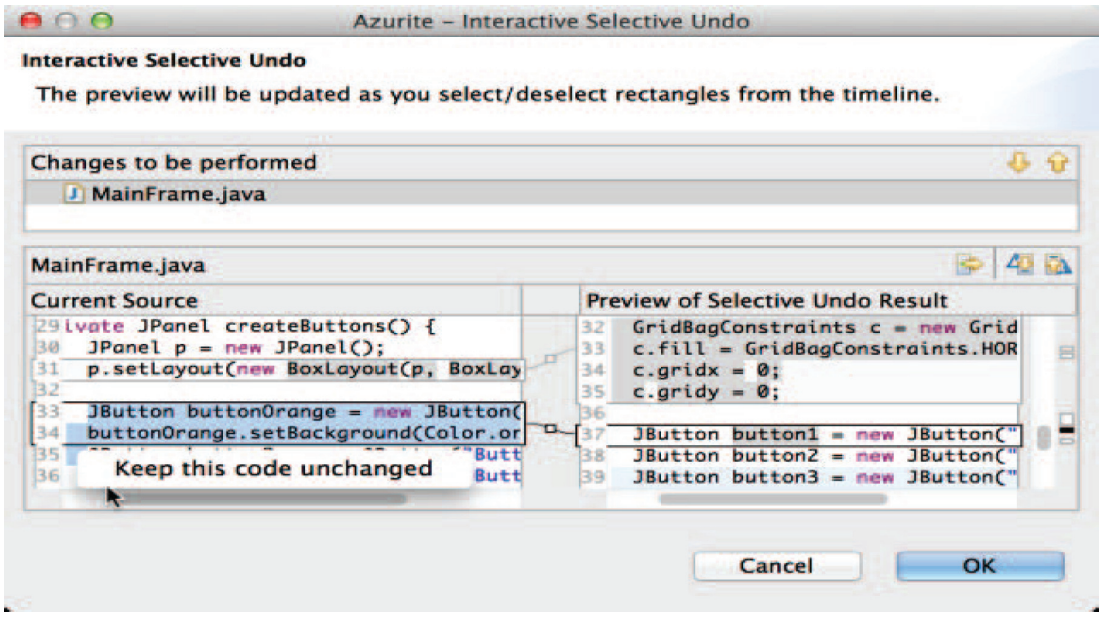


Figure 15.11 Azurite dialog that shows the *diff* between the current code (left) and the code in that region at a selected time in the past. Source: Fig. 1 of Yoon and Myers [2015].

command between B and C in Figure 15.7. The script model can be implemented by undoing all the operations back to before the operation to be omitted (using a conventional linear undo implementation), then skipping redoing the selected operation(s), and then redoing the subsequent ones. This could be expensive if the selected operation was far in the past or if the operations after it are slow to undo or redo.

One way to avoid the problems of Figure 15.7 is to do a *dependency analysis* to determine which past operations conflict. Then, the UI could either disallow selectively undoing operations that cause a conflict (such as preventing the selective undo of operation A in Figure 15.7), or the system could allow that operation to be undone and also undo any subsequent operations that depend on it (here, B, C, and D). This was formally investigated by Archer Jr. et al. [1984].

The script model was provided using a nice graphical interface in Kurlander's Chimera drawing program [Kurlander and Feiner 1988], where a *graphical history* window displayed the before-and-after of each operation (see Figure 15.8). Here, the user could select graphics in the History Panel for undoing or performing new operations in the past, and the system would propagate the edit forward by redoing subsequent operations. Chimera relied on the user seeing the consequences of their operations and editing accordingly, so it did not explore the issue of inconsistencies.

My group explored the script undo model in a research painting program called Aquamarine [Myers et al. 2015], which explored a variety of issues. We explored various UIs for allowing users to select which operations to undo, and the most popular and easiest to understand was to have checkboxes with each operation in the graphical history, which the user could uncheck to selectively undo that operation (see Figure 15.9). An alternative that was rejected was to have the undo operations themselves put into the graphical history (so the undo operations themselves could be undone), but this proved too confusing. Another interesting finding was that the users preferred that the thumbnail with each operation in the graphical history (Figure 15.9, bottom) displayed little or no surrounding context (and only showed the operation itself), in contrast to Chimera (Figure 15.8), where showing the context around each edit was said to be important.

15.8.3.3 Inverse Model

Another approach to selective undo is called *direct selective undo* [Berlage 1994] or the *inverse model*. Here, the inverse of the selected operation is performed and added to the current position in the undo stack. For example, consider selective undo of operation C in row 1 of Figure 15.12. Since this operation changed the color of the rectangle from green to blue, the selective undo of that operation is to

change it back to being green. This operation is added to the *end* of the undo stack (row 2a of Figure 15.12). Note that in this model the selective undo of operation C always changes the color to green (unlike the script model)—consider an alternative sequence of operations in row 1x—even though operation D' overrides the previous color of operation C, the selective undo of C still changes the color back to being green. This model also avoids the ambiguities of the script model with respect to deleted objects. As shown in row 2b of Figure 15.12, selective undo of a create is always a delete, which can be performed at any time where the object still exists without causing a problem for intervening operations.

With the inverse model, the designer gets to decide what the selective undo of an operation should do. Usually, it is obvious what the user would want—the inverse of changing a property like color or size is to change it back to what it was before (as in rows 2a and 1x of Figure 15.12); the inverse of create is delete (row 2b); the inverse of delete is to undelete the object and put it back, and so on. We have also decided that the selective undo of a move is to put it back where it was before the move happened, which is consistent with changing of other properties [Myers 1998b]; an alternative might be to move it back by the same *relative* amount in case the object is in a new position when the user invokes the selective undo. Selective undo of an operation would be disabled if the object was previously deleted. For example, after operation A' in row 2b, it is no longer possible to selectively undo B, C, or D since the rectangle is not on the screen.

Besides providing a clear meaning for selective undo, the inverse model provides other benefits. Since the selective undo operation is itself put onto the undo

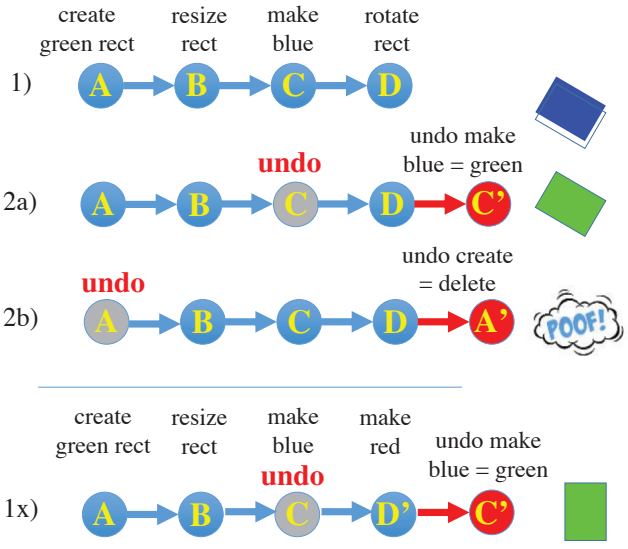


Figure 15.12 Example of selective undo using the inverse model. Source: Original by author.

stack, it can be undone using the regular Undo command if it does the wrong thing. Thus, hitting ^z after 2a or 2b would restore the screen to the way it was after D, and then redo would cause operation C' or A' to be redone, just like in the regular linear model.

The inverse model also enables *selective repeat*, which redoes the selected operation but using the currently selected object instead of the original object. For example, selectively repeating operation C would make the currently selected object be blue. The repeat operation would then be put onto the undo stack.

We explored the inverse model for text editing in the Azurite system [Yoon and Myers 2015]. The key issue here is the region problem as mentioned in Section 15.8.3.1—whereas with graphical objects in a drawing program, as shown in Figure 15.12, an object is either deleted or still there, with text many operations affect a *range* of characters, and it is not uncommon for some of those characters to have changed or been deleted, making the meaning of selectively undoing the operation unclear. In Azurite, we decided to do what we felt users were most likely to want when possible, and to ask the users what they wanted in less common edge cases. For example, if some characters had subsequently been deleted, selectively undoing a property change (like turning bold or changing font) would operate on the ones that were still there, and selectively undoing a delete would bring the characters back. However, consider a *replace* operation e_1 that changes the code from "myFontSize" to "myRectangleSize" and sometime later, another operation e_2 that changes it to "myRegionArea". The affected ranges of the two operations are overlapping and the "Rectangle" text inserted by e_1 is only partially available after both edits. In this case, it is not clear what the result of selectively undoing operation e_1 alone should be, so Azurite pops up a dialog box to ask the user [Yoon and Myers 2015].

15.9 The Back Button

The Web was originally designed for showing information, but with the rise of web *applications*, where users could fill in forms and submit them, a need for *undo* capabilities arose. However, there was little consistency of the UIs of web applications, with no standard menus or accelerator keys, and generally no Undo. The only similar operation was the *back* button, which users wanted to use to perform undo. For example, when booking a flight on an airline's web page, the user would pick the origin city on one page, then the destination city on the next page, and would want to go back to change the origin city (thereby undoing the submit of the origin city), but the original web applications would often crash when the user did this, and so they warned users not to use the back button at all. This was eventually fixed with more sophisticated web application programming, since now programmers can write JavaScript to detect when the user hits the back button. Web applications

have often supported a way to *cancel*—for example, by restarting the process with the site's home page; however, there is usually no easy way to cancel a purchase once it is in the user's shopping cart besides manually removing each item, or (for some sites) waiting a while for the cart to be cleared automatically.

As mentioned above in Section 15.3, Android provides a universal *back* button at the bottom of the screen, often represented as a left-pointing triangle (see Figure 3.27). This is supposed to go back to the previous screen for multi-screen apps, and thereby operates similarly to a web back button. However, it is also used to hide the on-screen keyboard and other purposes for various apps, making it unreliable for use as cancel or undo.

15.10 Multi-user Undo

As long as there have been systems that allow multiple users to collaborate synchronously (so that more than one user is editing a shared document at the same time), there has been a need for multi-user undo. Choudhary and Dewan back in 1995 articulated a set of requirements for multi-user undo, including (1) that it should be compatible with single-user undo, (2) that people should be able to undo *other's* edits, and (3) "semantic consistency" so that the undo does not mess up the application's model [Choudhary and Dewan 1995].[2] The simplest way to implement multi-user undo that meets these requirements is to require synchronous command execution across all users, but this may require locking out each user when another user is editing, which may be for a long time, especially in the presence of communication delays, as are common on the Internet. Therefore, Choudhary and Dewan provided an algorithm that relied on what we call *selective undo* (Section 15.8.3) so a remote command by a different user could be undone even if it was not the last command that was executed locally [Choudhary and Dewan 1995]. Since they wanted the effect of the undo to be as if the original operation had not happened, this essentially implemented the script model of selective undo (Section 15.8.3.2). Many researchers (e.g., Berlage and Genau [1993] and Ressel and Gunzenhäuser [1999]) and a few synchronous collaboration systems like Google Docs continue to investigate how multi-user undo should be designed and implemented, but it remains a problem.

2. Interestingly, their actual definition of "semantic consistency" is that "[t]he undo/redo model must ensure that the effect of undoing a command is equivalent to the state the system would be in if the undone command had never been executed, thus guaranteeing that the degree of consistency ensured by the concurrency control policy adopted by an application is not compromised by the use of undo/redo" [Choudhary and Dewan 1995]. This is not how even single-user undo/redo actually works, as explained in Section 15.6, so I argue that it is not an appropriate requirement for multi-user undo either.

Our PebblesDraw system (Figure 4.18) supported multiple people drawing at the same time who were collocated, so there was no network lag to worry about. PebblesDraw therefore provided both an Undo-by-User and a global Undo, so the users could choose whether to undo their last operation or the last operation on the application [Myers et al. 1998]. We implemented this using Amulet's selective-undo mechanism [Myers and Kosbie 1996], described in Section 15.8.3.3.

Another requirement that modern systems have (beyond those identified by Choudhary and Dewan) is: (4) that the undo mechanism must not unduly lock up and prevent editing for other users, who may be across the Internet. Interestingly, the Google Drive shared document editors for documents and drawings prioritize requirement #4 over the others, so undo can result in messed up content, especially if multiple people edit the same text at the same time and try to undo (see also Section 8.6.4 about multi-user text editing in general). Furthermore, modern systems like Google Docs do not support #2—people can only undo *their own* edits, so operations by others are only undoable or redoable by the person who originally did that operation. For example, consider the following sequence of edits on a shared text document by two people:

1. Person 1 types: MyCircleSize
2. Person 1 edits "Circle" to be "Rectangle", resulting in "MyRectangleSize"

   At this point, Person 1 can undo operations 1 and 2, but Person 2 cannot undo anything

3. Person 2 edits "ctangle" to be "gion", resulting in "MyRegionSize"

   At this point, Person 2 could undo the edit of "Rectangle" into "Region", but what should Person 1's undo do? As of August 17, 2022, Google Doc does the following:

4. Person 1 does Undo, resulting in "MyCirclegionSize"—replacing the "Re" that was left from Rectangle with the original "Circle".

There are similar interesting conflicts in shared *drawing* editors like Google Slides, which allows multiple people to edit graphical objects simultaneously. Consider the following sequence:

1. Person 1 creates a rectangle and makes it blue.
2. Person 1 edits the rectangle to be red.
3. Person 2 edits the rectangle to be green.

At this point, Person 2's undo will make the rectangle change back from green to red, but what should Person 1's undo do? As of August 17, 2022, Google Slides does the following:

4. Person 1 does Undo, and the rectangle changes from green to blue—the color before operation #2 that Person 1 is undoing.
5. Person 2 now does Undo, and the rectangle changes from blue to red, the color before Person 2's operation #3. In fact, no matter how the object would be edited by Person 1 or others (e.g., changed size, moved, changed borders), Person 2's undo would always turn it back to being red.

If either person deletes the rectangle, then the other person's undo doesn't do anything. Note that these are the same results as provided by the inverse selective undo model (Section 15.8.3.3) and arguably what each user would want. It is also *not* what would happen in the system of Choudhary and Dewan [1995] since it is *not* what would be the result if the original operation had not happened.

15.11 Scope of Undo

For applications that support multiple windows operating on different files, an interesting question is whether the undo stack for a user is separate for the different windows? Interestingly, Microsoft Office (at least as of v2016) is *inconsistent* with respect to this issue—each window in Microsoft Word is completely independent, whereas for Microsoft Excel there is one undo stack *across* all windows. That is, for Word, if I make edits to file1.docx and then to file2.docx in different windows, if I then put the cursor into file1.docx, Ctrl-Z will only undo operations on file1. However, with Excel, hitting Ctrl-Z jumps from file to file in reverse order of the operations in *any* window, unless the window has been closed. Most other applications are like Word, so they have a separate undo list per window. Of course, separate applications always have independent undo lists.

Another scope issue is whether the undo stack is saved with the file, and therefore available if the file is closed and later reopened. I am not aware of any programs that do this, however—all applications start the undo list as empty whenever a file is opened.

15.12 Implementing Undo and the Command Object Pattern

Although most of this book's discussion about how to *implement* interaction techniques in software is in Chapter 20, it seems useful to put the implementation of Undo/Redo/Repeat using *command objects* here since it is focused solely on these interaction techniques.

Undo/Redo/Repeat can be implemented in a variety of ways, using arbitrary code or data structures. In object-oriented design, the most common way is to use the *command object pattern*, which is a way to implement operations in an object-oriented system. Here, all command objects have a standard interface or method for invoking the command, often called *do* or *execute*, so that no matter the operation it can be invoked the same way. For example, a command object might be assigned to each menu item and button in an interface, and then the user selecting that item can cause the widget to call the execute method on the command object, without knowing anything about what it will do. The relevance to this chapter is that the command object can also have other methods that support undo, repeat, selective undo, and so on and thus provide a standard way for those operations to be invoked as well. Since these operations are not always allowed, often there are further methods that return Booleans if the operation is currently allowed, which might check, for example, for a delete operation whether anything is selected. Similarly, an `undo_allowed` method might check whether this command can be undone.

The command object pattern is one of the 23 patterns in the original 1994 *Design Patterns* book [Gamma et al. 1995], but the use of command objects dates from before that. MacApp, one of the first object-oriented UI frameworks, from around 1986, provided command objects that supported undo as one of its six major classes [Schmucker 1986]. UniDraw, an early C++ research toolkit from 1989, also supported the command object pattern [Vlissides and Linton 1989]. The Gina multiuser undo system [Berlage and Genau 1993] and its selective undo mechanism were both implemented using command objects [Berlage 1994].

My Amulet system extended the command object pattern to support *hierarchical command objects* [Myers and Kosbie 1996], which meant that lower-level parts of the software, such as the widgets themselves, could use command objects for their internal operation (e.g., the menu widget behavior was itself executed by the widget's command object), which then called higher-level command objects when invoked (e.g., the menu widget's command object would invoke the command object associated with the selected item).

The Amulet command objects provided methods that the developer implemented for *do*, *undo*, and *redo*, and also a variety of *selective-undo/-redo/-repeat* methods. There was also a full set of query methods to determine whether the operation was allowed in the current context. For example, as explained in Section 15.8.3.3, selective repeat performed the operation on the current selected item, so the `selective_repeat_allowed` method would check that there was something selected now and that the operation made sense for that object (e.g., a change-font operation would not be allowed on a rectangle). The Amulet model

also provided for all the information needed to undo or redo the operation to be stored in the command objects, such as the old and new values that were used, and the object originally operated on [Myers and Kosbie 1996].

15.13 Macro Recording and Programming by Example

Macro recording is when the user can go into *record* mode, perform operations with a program, and stop the recording, and then those operations are recorded into a program, also called the *macro* or *script*. This program can later be executed to perform the operations again. Typically, a macro can be recorded by end-users who do not necessarily know anything about programming. The idea of creating programs using macro recording dates back to at least the 1970s with the Emacs editor [Stallman 1979], whose name actually stands for Editor MACroS. Other editors like Microsoft Word and spreadsheet programs also support macro recording.

The reason that macro recording is included in this chapter is that it is very easy to provide recording if the command object model (Section 15.12) is used to implement undo, especially if selective repeat (Section 15.8.3.3) is supported. As we showed in my Topaz system [Myers 1998b], executing selective repeat of a selected series of operations (Figure 15.10(a)) will execute the macro again, and various options for changing the parameters of the operation, such as using the current selection or the most recently created object instead of the original object used, can make the macro more useful. For example, Topaz allowed users to write macros to create a recursive Sierpiński Gasket and De Morgan's law (Figure 15.10(b) and (c)).

In other systems, like Microsoft Office, all commands are represented by back-end library function calls, so the macro is collected as a program containing these function calls with the appropriate parameters, and a knowledgeable user can then edit the code of the macro.

Researchers have studied macro recording for many years, and it is often called *programming by example* (PBE) or *programming by demonstration* (PBD) [Myers 1986a, Cypher et al. 1993, Lieberman 2001] since the program is created by giving an example or demonstration of what is supposed to happen. One key difficulty with such systems is dealing with the values of parameters, especially for what is being operated on by the command, which is called the *data description problem* [Li et al. 2018]. When the macro is used later, what values should be used—the same constants as when demonstrated, or different values computed or found in the run-time context? Other difficulties include dealing with control structures, like loops and conditionals, since the example provided by the user is likely to be a straight-line program doing the desired actions in a particular context, but the run-time

program may need to do different operations depending on the new run-time context, or loop a different number of times. Approaches to help the system address these issues have included requiring the user to give *multiple* examples of what should happen in different situations [Frank and Foley 1993, McDaniel and Myers 1998, Scaffidi et al. 2008], specifying the desired values through dialog boxes *during* the initial demonstration [Myers et al. 1989, Halbert 1993], generating a program and requiring the user to edit it *afterward* [Myers et al. 1993, Wolber 1997], and using natural language (speech) to explain the required generalizations [Li et al. 2017, 2018, 2019]. Another dimension is the level of machine learning or AI that is used to *infer* what the user wants, versus requiring the user to directly specify the generalization [Myers et al. 2000c].

15.14 Summary of Considerations for Undo/Redo

This chapter has explained the large number of different types and options for supporting undo and related methods in applications. Here is a quick summary:

- *Cancel*: Which operations can be canceled while in progress, and how? Typically, all dialog boxes should have a "Cancel" button, and interactive operations (like dragging) should support hitting some key (e.g., the Esc key) to abort them while in progress. Stateful web and mobile applications should support easy and clear ways to start over, empty the shopping cart, and/or get back to the original starting state.
- *Which undo model*: Pretty much all desktop systems today support the linear undo model, but for mobile and web applications it is unusual to support undo at all, but some support single-level undo or multi-level undo for some operations. Other undo models (like selective undo) are still rarely found outside of research systems. However, for advanced systems for expert use, especially editors that are supposed to foster creative exploration, such as Photoshop and some code editors, providing other kinds of undo may help, such as selective undo or region undo.
    - *Multiple Users*: If there needs to be support for multiple users, then other undo models will be required since the regular linear model does not work in the presence of multiple users' independent edits on shared content.
- *User interface*: On regular computers, Ctrl- (or Command) Z is always Undo, but there are still inconsistencies over the keystroke for redo (Ctrl-Y or Ctrl-Shift-Z). On systems without a keyboard, such as mobile devices, it is harder to have undo support, and there is no consistency on its UI. For

more advanced undo models, such as selective undo, there must be a way to understandably represent past commands so the user can select them.

- *What can be undone*: There are likely to always be operations that cannot be undone, such as sending an email, printing, or reformatting a disk. These should be minimized as much as possible, and good UI design practices suggest making sure that users do not perform these operations by accident. If using a linear undo model, it is often better to allow the user to undo beyond such commands, to continue to undo operations done before the non-undoable command.
- *Unit of undo*. If there are many low-level operations performed quickly, such as typing text, deleting using auto-repeating backspaces, or moving incrementally using arrow keys, it may make sense to coalesce these into a single undo. Typically, it requires user studies to understand what is most desirable and usable.
- *Across or within windows*: A scoping issue with undo is whether the undo crosses windows (like Excel) or stays entirely within the active window (like most other programs, including Word).

15.15 Chapter Conclusions

Providing the ability to undo and cancel operations is fundamental to supporting a great user experience, as has been known since the 1970s. Although desktop applications have uniformly supported undo since at least the 1980s, usually using the linear model for undo, very few mobile or web applications support undo at all, and generally require the user to manually reverse the effects of partial operations (like removing items one-by-one from a shopping cart) rather than supporting cancel or undo. And the coming "Internet of Things" (IoT) and smart appliances virtually never provide an undo operation. As users move more of their activity away from desktop applications, it will be important to investigate how to support undo and cancel across all of the user's activities and devices.

CHAPTER 16

Interaction Techniques for Help

16.1 Introduction

Given the importance of *helping* the user learn and effectively use technology, and the prominence of "help" in guidelines (e.g., number 10 of Nielsen's heuristics is "Help and documentation" [Nielsen and Molich 1990]), it is somewhat surprising how little research and innovation there has been around interaction techniques for help. Of course, the topic of *documentation* is well-researched, and there are whole books and conference series devoted to how best to write and present documentation (e.g., ACM SIGDOC), but the content of documentation and manuals do not count as *interaction techniques*, so they are not covered in this book. The *structure* of the documentation may be considered an interaction technique when it is interesting. For example, as mentioned in Section 2.5.1.9, one of the motivations for the early development of hypertext was to make documentation easier to use and navigate since it often contains terms that are explained elsewhere.

It may actually not be surprising that there are few interaction techniques for help given that every interaction technique is a new thing that users need to learn, and when users are stuck trying to learn how do something with an application, they do not then want to have to learn how to use the help system, especially if it has novel interaction techniques! In fact, one of the "usability slogans" in Jacob Nielsen's classic book is that "help doesn't":

> **Help Doesn't**
> "Sometimes, online help and documentation doesn't really help the users. That is to say, users often do not find the information they want in the mass of possible help and documentation and, even if they do find it, they may misinterpret the help. Also, help adds an extra set of features to a system, thus complicating the interface just by virtue of existing. ...It is always better if users can operate the system without having to refer to a help system ...so

a user-hostile interface does not get user-friendly even by the addition of a brilliant help system". [Nielsen 1993, p. 16].

Nevertheless, there are interaction techniques relevant to helping users. Covered in this chapter are the *status bar* of the screen or windows, popup *tooltips* with short descriptions of what is being pointed at, *autocomplete* to help people know what to enter, and *help modes* for querying about items or limiting the interactions available.

Other interaction techniques that might be considered "help" are covered elsewhere in this book. For example, spell-checking (which helps users enter their content) was covered in Section 8.4.2, and intelligent agents (Chapter 18) have often been used to provide help.

16.2 Status Bar

One of the earliest interaction techniques relevant to help is the *status bar*, or *status line*, which is a row of text, typically at the bottom of the screen or window, in which applications and the system can display messages, typically about the status of the system or application. When the main displays that people were using were 24 by 80 terminals (see Section 2.3 and Figure 2.7), there were commands to move the cursor to the bottom of the screen so that applications could use that area for messages. Some displays even added a 25th row specifically for the status bar. The WordStar application was unusual in using the top row of the terminal as the status bar, instead of the bottom one, because it wanted the labels for the function keys to be at the bottom closer to the physical keys (Figure 5.6(a)).

When window systems started to be used, the bottom of each window was typically used for the horizontal scrollbar, but even as early as Windows 1.0 in 1985, applications could take part of that row for the status bar. For example, the rightmost window in Figure 2.17 shows "Page 1" as the status. More typical is for the status bar to be the full width of the window, below the horizontal scrollbar, as in the Microsoft Word status bar in Figure 16.1 or the Emacs status bar (shown in gray) at the bottom of Figure 15.6. The bars across the top of the Apple (Figure 3.26) and Android (Figure 3.27) phones and tablets are also "status bars" since they show the

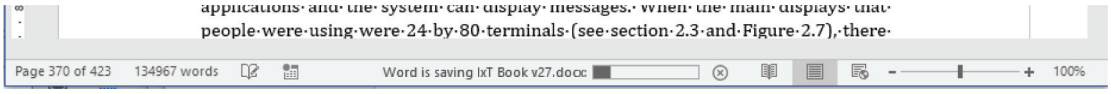


Figure 16.1 Microsoft Word's status line while saving this book. Source: Screenshot of Microsoft Word 2016, used with permission from Microsoft.

time, the status of the phone and Wi-Fi connections, the battery charge, whether an alarm is set, and so on.

There are a wide variety of kinds of content displayed in the status bar. System uses have included showing the state of keyboard modes, such as Insert Mode or CapsLock, and global information such as the device state and battery charge. Some older systems displayed information in the status bar about whatever the mouse cursor was hovering over, such as menu items or iconic buttons—later this information moved to be shown in tooltips (Section 16.3).

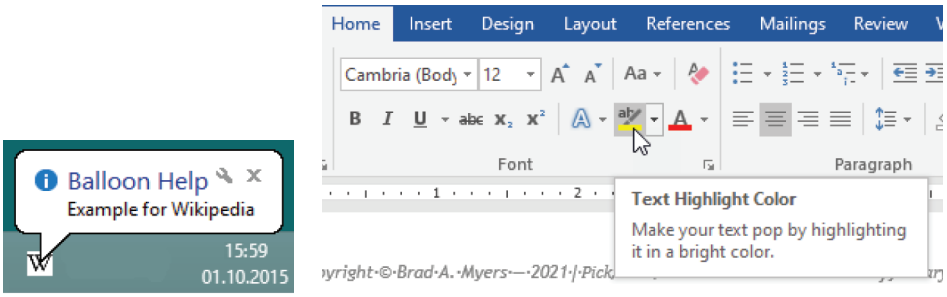
Application-specific status bars have displayed the current page of the document, the current type of view of the document, the zoom factor, and so on. Figure 16.1 shows that the progress bar for saving the file is also shown in the Word status bar. Sometimes the items in the status bar serve as buttons to allow the status to be changed. For example, clicking on the word count at the left of Figure 16.1 brings up a dialog box with other statistics, and the slider and buttons on the right of Figure 16.1 can be used to change the zoom factor of the current page.

16.3 Tooltips

A *tooltip* is an interaction technique where a small information box pops up when the cursor is held over an item and disappears when the user moves away. On touchscreens without a mouse cursor, sometimes tooltips appear on long-press. Tooltips have also been called *ScreenTips*, *Balloon Help* on the Macintosh, or a *hoverbox*.[1]

I haven't been able to find the origin of the word "tooltip". Wikipedia as of January 2021 says that the term comes from older Microsoft applications (e.g., Microsoft Word 95), but one author reported it was being investigated for Excel v.3.0, which dates from about 1990—Microsoft performed a study about how to make the new iconic toolbars of Excel more understandable to users and found that tooltips were helpful [Dye and Graham 1995]. Tooltips were more ubiquitously available in the Windows toolkit by Windows 95. Meanwhile, Apple was experimenting with various ways to improve the understandability of the Macintosh UI, and they introduced what was called *Balloon Help* in the 1991 release of System 7.0, although a patent on Balloon Help was filed by Apple's Taligent subsidiary in 1992 [Matheny et al. 1992]. The "Balloon Help" name was chosen since the information popups appear above the item, using a "*speech bubble*" shape from paper comic strips (Figure 16.2(a)). Modern tooltips (Figure 16.2(b)) look and behave very similar to the original ones from Excel—hanging below the item of interest, appearing

1. Wikipedia (as of January 2021) classifies a "hoverbox" as different from a tooltip in that a hoverbox is larger and supports formatted text. I disagree that this is an important distinction and classify all small information boxes that pop up on hover or long-press as "tooltips".

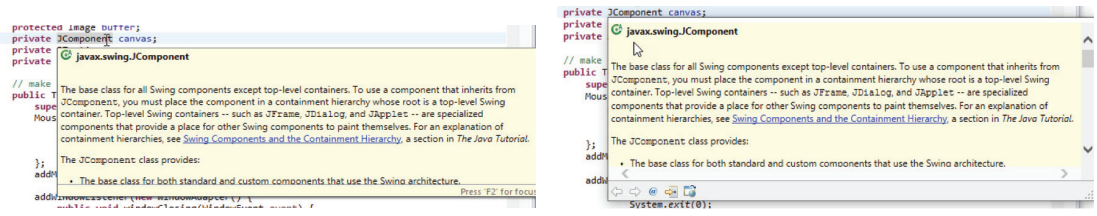


(a) (b)

Figure 16.2 (a) Balloon Help on the Macintosh. (b) Tooltips in Microsoft Office. Sources: (a) By Phrontis, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=44033371. (b) Screen capture of Microsoft Word 2016, used with permission by Microsoft.

only when the cursor is held still for a period of time, and disappearing if the user moves the cursor off the item.

An interesting variant on tooltips was provided by the Eclipse Integrated Development Environment (IDE), from around 2002, which it originally called *infopops* [Halsted and Roberts 2002]—users could move their mouse from the item of interest *into* the tooltip, which was then converted into a regular popup window that could be interacted with. For example, the tooltips often contained links to more detailed information or contained longer content that might need to be scrolled (Figure 16.3). Note that with regular tooltips it is not possible to move the cursor into the tooltip since as soon as the cursor leaves the item of interest the tooltip will disappear, and often there is a gap between the item and the tooltip (Figure 16.2). Therefore, Eclipse detected if the user was moving in the direction of the tooltip, and it added a delay before the tooltip was removed after the cursor left the item of interest. Another distinction was that Eclipse provided these tooltips for most of the content inside the windows, whereas Balloon Help and Microsoft's



(a) (b)

Figure 16.3 In Eclipse, tooltips that appear when users (a) hover over an item can be turned into (b) regular windows where users can scroll and click on items. Sources: Screen captures of Eclipse v. Neon.3 Release (4.6.3) taken in January 2021.

tooltips were mainly for the widgets and controls *around* the main content. Providing tooltips for the contents of web pages dates back to at least 1996, when Jakob Nielsen [1996] submitted a patent on a way to put the tooltip information into the HTML; today, the HTML "`title`" attribute will appear in tooltips.

16.4 Autocomplete as Help

As discussed in Section 8.4, autocomplete is often used as a means to reduce the amount of typing needed. However, we have shown [Mooty et al. 2010, Santos and Myers 2017] that it is also a popular means for *getting help* about what is available and recommendations about what to do next. As presented in Section 8.4.3 and shown in Figure 8.24, when the user types a dot "." after an item, the editor will show a list of items that can follow the original item, thereby helping users understand what is allowed. If the items have helpful names, this may be sufficient for the user to know what to do. Sometimes, as in Figure 8.24, if the user hovers over an entry in the list of items, a tooltip will pop up to provide a brief description of that item. Research suggests that this often is programmers' *preferred* way to explore what is available in a library since it is directly in the editor, so it is very quick and avoids needing to switch contexts to the documentation or a browser window.

In web browsers' search bar (or address bar) and other text-entry fields, autocomplete is also available after the user types a few characters. This can help users know what makes sense or is popular to type next. For example, in Figure 8.26 the user can see what is popular to compare with trains, and in Figure 8.25, the user can explore the list of available airports that American Airlines flies to.

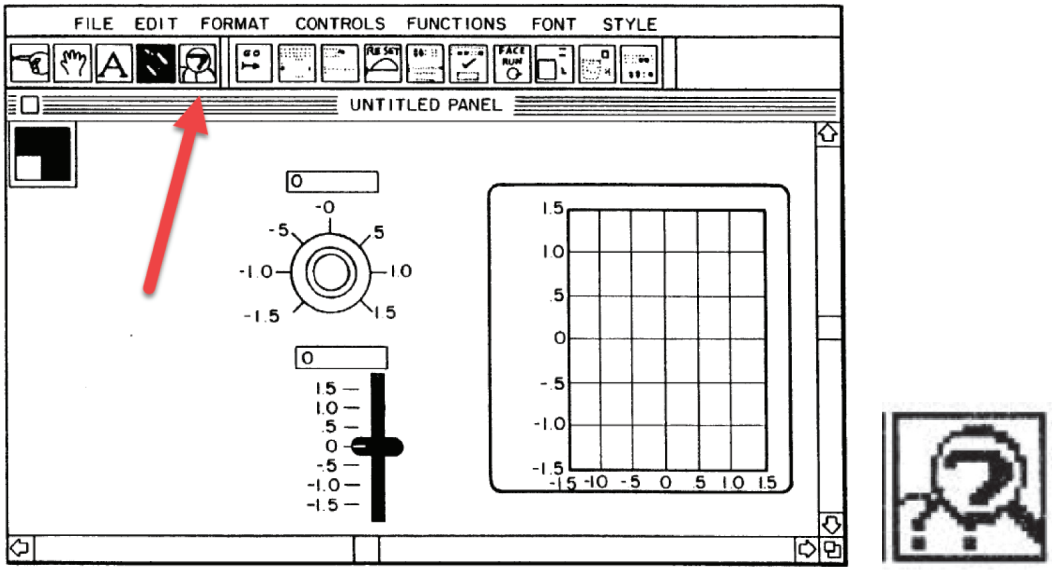
16.5 Help Modes

16.5.1 Ask About a Screen Item

A number of systems have allowed the user to go into a special mode and click on controls in the interface to get help on them. This was available, for example, in the first version of LabVIEW from 1986 [Kodosky et al. 1986]. When the user clicked on the "?" icon at the top of the LabVIEW window (Figure 16.4), the system went into "help" mode, and it showed documentation about the next item that was clicked on.

Many Windows dialog boxes had a "?" icon that worked in the same way, as did the "What's This" menu item in Windows 95, with the accelerator key of Shift-F1 (Figure 16.5). Typically, the user could then click on widgets and icons in the UI.

We generalized this idea in a research system called Crystal from 2006, where the user could click on the "*Ask about a location...*" menu item, which put the system into a mode where the next click would drop a pink question mark on the window, and the user could then ask specific questions about that location (Figure 16.6). (Or



(a) (b)

Figure 16.4 Clicking on the 5th icon in LabVIEW (detail at right) goes into help mode. Source: Fig. 5 of LabVIEW patent [Kodosky et al. 1986].

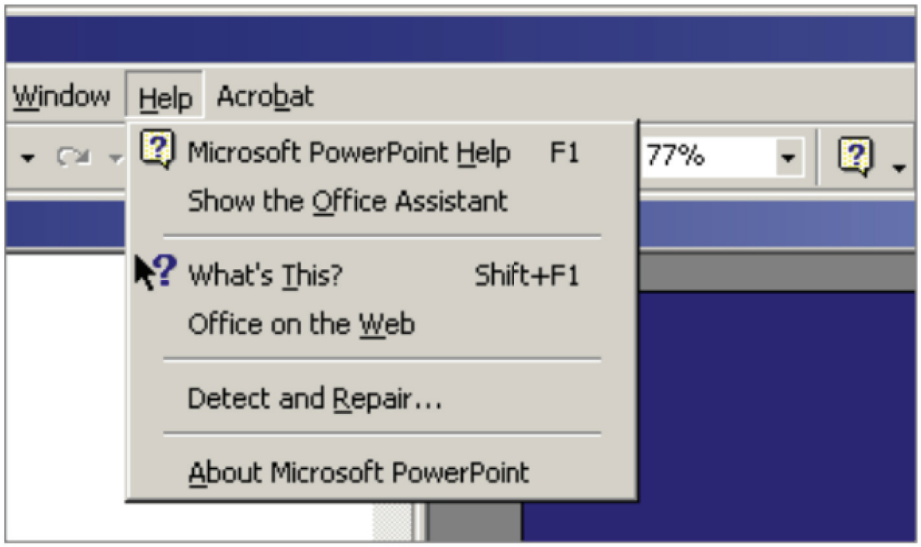


Figure 16.5 PowerPoint 1995 help menu, containing "What's this?". Source: Screenshot of Microsoft Powerpoint 1995, used with permission from Microsoft.

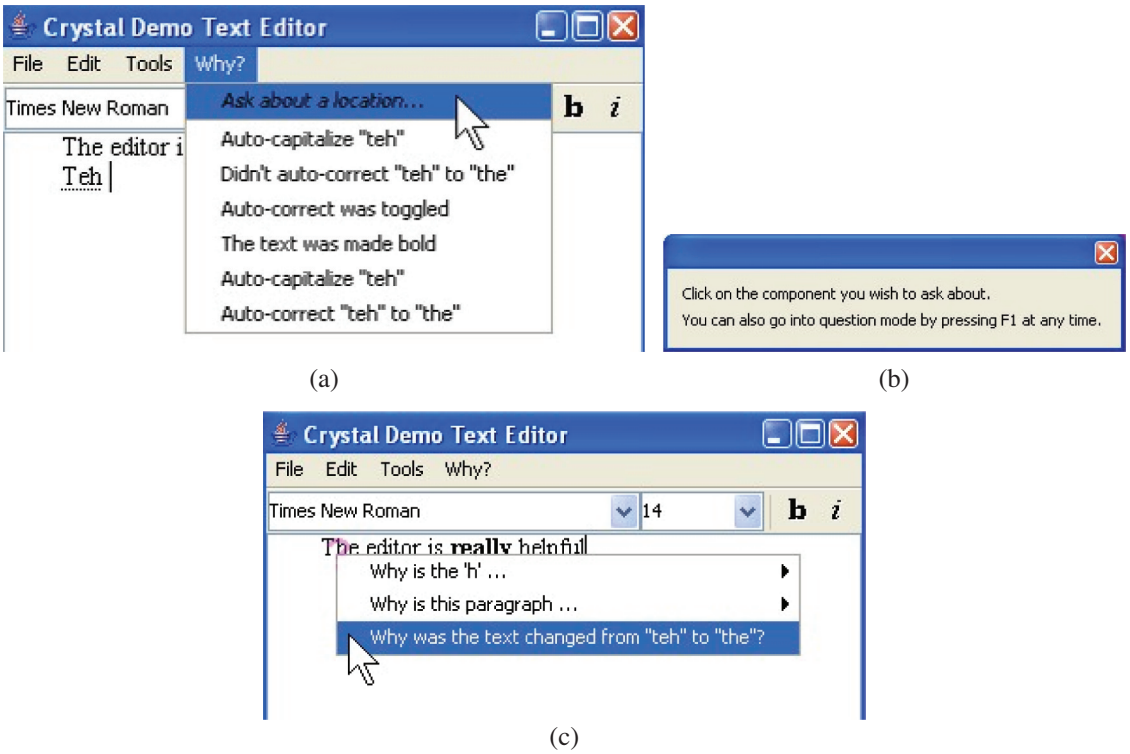


Figure 16.6 (a) When the user asks about a location in Crystal, the system goes into a mode (b), where the next click drops a pink question mark (c) and pops up a menu of automatically generated questions about what happened there. Source: Screenshots used in talk about Myers et al. [2006].

the user could put the mouse cursor at a spot and hit F1.) The question mark could be put anywhere in the content (e.g., on a word or whitespace in the text editor), and Crystal would automatically generate explanations about why that word or whitespace was formatted the way that it was, which often involved pulling properties from multiple styles and previous user actions [Myers et al. 2006].

These kinds of moded help systems are not currently popular. I speculate that most of the need for them has been removed due to more ubiquitous use of tooltips, which have better discoverability and usability, and a general sentiment against modes (see Section 23.8), especially for situations aimed at novices. Although we showed with Crystal how to generalize help to application-specific content without significant extra implementation or UI burden [Myers et al. 2006], that hasn't caught on either.

16.5.2 Stencils

Another mode-based way to provide help is to change the UI of the actual application in synchrony with the steps of a tutorial. For example, Windows 95 had a tutorial mode to show users how to do various tasks step-by-step. If the user clicked on the wrong widget, then Windows beeped and eventually pointed to the correct item to use on the screen. This worked better than having to refer to documentation even when it contained pictures since users could have a hard time matching the pictures to what was on the screen [Kelleher and Pausch 2005].

Research showed that sometimes users still had trouble focusing on the correct part of the interface, so the *Stencils* idea was to cover up most of the interface and have a cut-out of the part that the user should pay attention to right now [Kelleher and Pausch 2005]. For example, in Figure 16.7, the stencil is the blue translucent overlay, and the user should click on the "trevor" button. Additional annotations

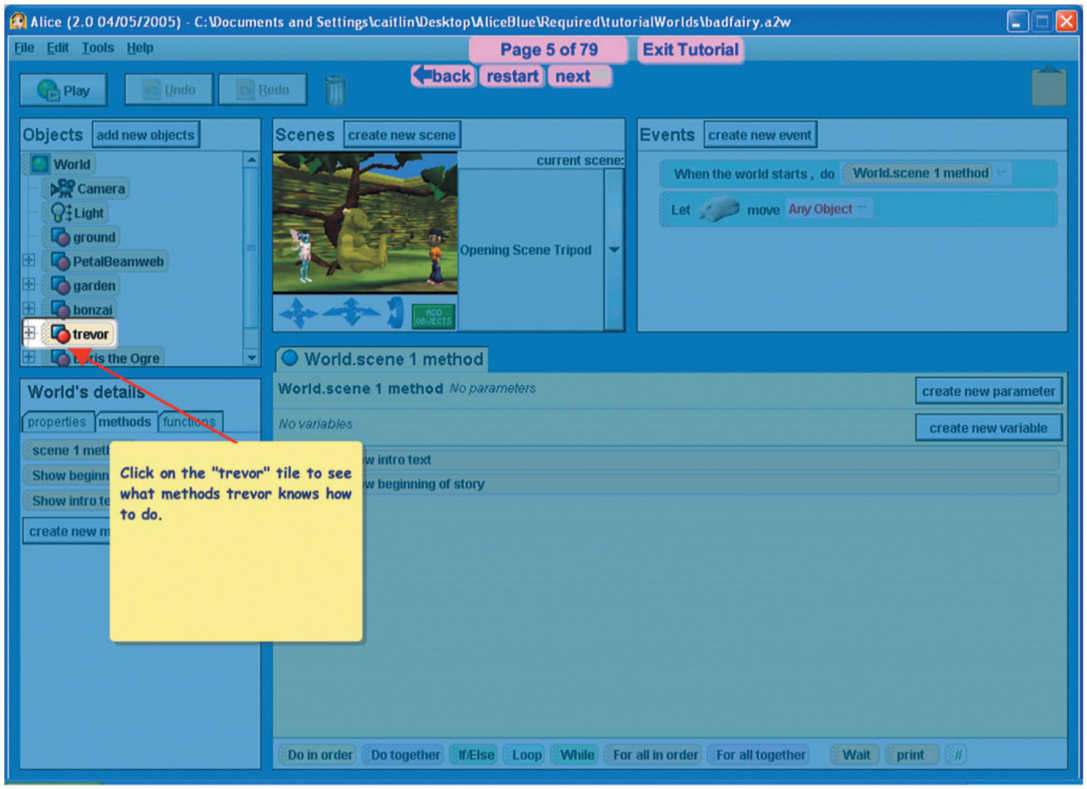


Figure 16.7 Blue Stencil covers the screen except for the "trevor" button that the user should click on. The pink text at the top tells the tutorial step, and the yellow note explains what to do in this step. Source: Courtesy of Caitlin Kelleher, used with permission.

could be put on top of the stencil, such as the pink text at the top and the yellow note. In a user study, the Stencils approach resulted in the same level of learning in a substantially shorter period of time, with fewer errors [Kelleher and Pausch 2005]. However, I am not aware of any widely available application that has used this technique.

16.6 Considerations for Help and Chapter Conclusions

While some interaction techniques for help such as tooltips and autocomplete are now ubiquitous (and should be used by application developers), other interaction techniques have not caught on and should be approached with caution. After all, when a user needs help with an application, they don't want to have to learn a new interaction technique just to access the help, and having to do so likely would prevent some portion of the users from finding out what they need to know.

Probably more research is needed on how to provide help on mobile devices since techniques like tooltips and using the status bar do not transfer well. In the future, as intelligent agents (Chapter 18) become more successful, future help systems will allow users to ask their questions using natural language, which may be able to leverage some of these interaction techniques, such as allowing users to point to what they want information about (Section 16.5.1) or allowing the agent to focus the user's attention on part of the screen (Section 16.5.2).

CHAPTER 17

Interaction Techniques for People with Disabilities

17.1 Introduction

The previous sections have mostly focused on interaction techniques that have been designed for people with average abilities. However, for many people who are disabled, those designs do not work well or at all. Globally, estimates are that 1 billion people or 15% of the world's population experience disabilities, and of these, between 110 and 190 million people (2% of the world's population) have severe or extreme difficulties in functioning [Kostanjsek et al. 2013]. In the US, the National Center on Birth Defects and Developmental Disabilities, Centers for Disease Control and Prevention, reports that 61 million adults in the United States live with a disability, which is about 26% [CDC 2020a]. Of these:

- 13.7% of people with a disability have a mobility disability. These people may have difficulty using a mouse or a conventional keyboard.
- 10.8% of people with a disability have a cognitive disability with serious difficulty concentrating, remembering, or making decisions. These people may have difficulty with the complexities of user interfaces and their interaction techniques.
- 5.9% of people with a disability are deaf or have serious difficulty hearing. These people can often use regular GUIs and IxTs, but may need captions for videos or speech interactions.
- 4.6% of people with a disability have a vision disability that includes blindness or serious difficulty seeing even when wearing glasses. These people may not be able to use graphical interfaces at all without adaptations.
- 40% of adults aged 65 years and older have a disability [CDC 2020a].

This chapter discusses adaptations and designs for interaction techniques to make them more *accessible* to those people, which means that the barriers are reduced or removed for people to successfully understand and interact with them. While the definition of "disability" varies, and some people who meet that definition may be able to use conventional interaction techniques, there is clearly a large segment who may benefit from making IxTs more accessible.

Research on assistive and accessible interfaces has a long history. For example, Gregg Vanderheiden was one of the founders of the Trace Center in 1971 at the University of Wisconsin–Madison to address the communication needs of people who are non-speaking and have severe disabilities [Trace 2021], and the Trace Center continues to be a major contributor to accessibility. Accessibility is also the focus of the annual International ACM SIGACCESS Conference on Computers and Accessibility (abbreviated ASSETS) that started in 1998. Virtually all modern operating systems now have built-in features to help with accessibility, but more work is always needed especially as new modalities become popular, such as virtual reality/augmented reality (VR/AR) (Chapter 12) and artificial intelligence in interfaces (Chapter 18).

A key challenge is that people may have different *combinations* of disabilities (such as both motion disabilities and blindness), and each individual may be disabled to different degrees, which may change over time. For example, some diseases are progressive, with individuals getting more disabled over time (such as amyotrophic lateral sclerosis (ALS), or even old age, see Section 17.3), whereas others are more stable or may decrease in severity (such as stroke or spinal cord injury). Especially challenging (and less studied) for the IxT designer are cognitive disabilities since many of the adaptations discussed in this chapter actually make the overall interface *more* complex, such as adaptations for low vision, which require the users to memorize dozens of keyboard commands as well as the structure of the user interface so they can navigate it.

In addition to there being a large need for accessible interfaces, there are standards and laws that require it. For example, in the US there have been laws dating back to 1986 that require better access to electronic equipment in the federal workplace [Glinert and York 1992]. The Trace Center contributed to ISO 9241-171:2008, "Ergonomics of human-system interaction—Part 171: Guidance on software accessibility," published in 2008, which "provides ergonomics guidance and specifications for the design of accessible software for people with the widest range of physical, sensory and cognitive abilities, including those who are temporarily disabled, and the elderly" [ISO 2008]. There are also laws in the US, the European Union, and elsewhere that require websites and smartphone apps to be accessible, and people have won lawsuits against companies and government agencies that have inaccessible websites.

17.2 Related Concepts and Definitions

Interfaces created for people with disabilities are often called *accessible interfaces* [Keates et al. 1998] or *assistive interfaces* or *inclusive design* [Keates et al. 2000]. *Assistive technology* includes assistive user interfaces but also any other piece of equipment or product that makes the item more accessible. For example, there is a strand of research on 3D printing of custom holders for spoons or bows for a cello that allow them to be held by people who cannot make a fist or may be missing a hand [Buehler et al. 2014], which qualify as assistive technology. However, since many of these technologies are not interaction techniques, they are not discussed further in this chapter.

Universal design means that one design works for everybody regardless of age or abilities [Mace et al. 1991]. The classic example is curb cuts, which obviously benefit people in wheelchairs, but are also easier for people with rolling luggage or baby strollers, those who have joint pain going up stairs, or bicycle riders. This is as opposed to accessibility features where disabled people use a *different* interface than others.

Situational impairments are when an otherwise abled person is in a situation that makes it difficult to use regular interfaces, such as when needing physical stability, high tactility, increased visibility, and increased comfort [Silfverberg 2003]. For example, when riding on a subway and needing to hold on so that only one hand is free, or while driving so one cannot look at the interface, at least not for more than 2s (the maximum amount of time to look away from the road according to the US National Highway Traffic and Safety Administration [Taub 2018]). In these cases, interfaces that are designed for people who are disabled may benefit others who are temporarily disabled due to their current situation.

17.3 Kinds and Causes of Disabilities

There are a variety of disorders and conditions that can affect people's abilities to use interaction techniques; the following are some of the most notable.

17.3.1 Muscular Problems

When people have reduced or no control over their muscles, then mouse or touch-based interactions can be difficult or impossible. These disorders may also affect the use of conventional keyboards and even buttons. Sometimes speech is also affected. Here are some causes:

- *Cerebral palsy* (CP) is a disorder that is caused by brain damage to the motor control parts of the brain, often prenatally or as an infant, and results in limited physical control. Symptoms sometimes include not being able to talk or control their hands, but often CP has no effect on intelligence, and there is a

wide range of abilities. Unlike some other disorders, people with CP usually do not get worse. The prevalence is about 2.11 per 1,000 live births worldwide [Oskoui et al. 2013].

- *Amyotrophic lateral sclerosis (ALS)*, also called *Lou Gehrig's disease*, is a disorder resulting in muscle weakness and atrophy throughout the body due to the degeneration of motor neurons. It is a degenerative disease, meaning it gets worse over time, with the majority of ALS patients dying within 2–5 years of receiving a diagnosis [Mehta et al. 2018]. However, Stephen Hawking, a famous theoretical physicist (Figure 17.1), lived to be 76 years old after being diagnosed with ALS in his twenties. Most people with ALS lose the ability to speak and move, and eventually they may lose the ability to eat and

Figure 17.1 Stephen Hawking, a noted physicist with ALS who used an augmentative and alternative communication (AAC) device. Source: By NASA/Paul Alers, Public Domain, https://commons.wikimedia.org/w/index.php?curid=16350790.

even breathe on their own. The prevalence of ALS is about 5.0 per 100,000 population in the US [Mehta et al. 2018].

- *Parkinson's disease* is a slow-acting degenerative disorder of the central nervous system that most famously causes tremors in the hands but also causes rigidity, slowness of movement, and difficulty with walking. It usually affects older people, often after age 60. Prevalence is about 6.2 million people worldwide in 2015 [GBD 2016].
- *Muscular dystrophy (MD)* also causes loss of motor control but is different from the above disorders in that MD causes loss of gross motor control and strength while retaining fine motor control, at least for a while. It often starts at birth and is more prevalent with boys than girls. The estimated prevalence of the two most common forms of MD is 1 in every 7,250 boys aged 5–24 years in the US [Romitti et al. 2015].
- *Spinal cord injury (SPI)*, depending on where it is, can cause loss of use of the legs, which typically has little impact on the use of the interaction techniques described in this book, but injuries higher on the spine can affect the use of the arms and hands. The prevalence of SPI in the United States is that there are approximately 270,000 people in the US with SPI, of whom about 16% are completely without use of either arms or legs (quadriplegic) [NSCISC 2012].
- *Repetitive stress injury* (*RSI*—also called repetitive *strain* injury), *carpal tunnel syndrome*, and *tendonitis* are all disorders that can make it painful or impossible to use body parts in the usual way. These disorders are relevant to interaction techniques mainly when they affect the use of the hands, arms, and fingers for the mouse, keyboard, or touchscreens. Often, these disorders are caused by repetitive actions (like the frequent need to push the mouse buttons or move the mouse scroll wheel) or general overuse, and these may cause people to switch to alternative input methods to reduce the need to perform the same actions. For example, I have needed to switch from using a mouse to using a trackball instead (see Figure 17.2) due to significant tendonitis in my right index finger from overuse of the mouse button and scroll wheel.
- *Just being elderly* also contributes to reduced physical dexterity and endurance, lower eyesight, reduced cognitive abilities, and so on.

17.3.2 Visual Impairment and Blindness

Visual or *vision impairment*, also called *visual* or *vision disability*, is a decreased ability to see to a degree that causes problems not fixable by usual means, such as

Figure 17.2 The Kensington Orbit® Trackball with Scroll Ring that I used to write this book. Note that one can use different fingers to move the ball and gray scroll ring, and the thumb is used to press the left button, which allows the index finger not to be used. Source: Image taken by author.

glasses or contact lenses. *Blindness* is complete or near complete loss of vision. Visual impairments globally are caused by lack of access to glasses or contacts, cataracts (where the lens of the eye becomes cloudy), glaucoma and other eye diseases (where there is damage to the optic nerve), injury to the eye or brain, diabetes, and a wide variety of other diseases and disorders. The World Health Organization reports that, globally, at least 2.2 billion people have a vision impairment or blindness, of whom at least 1 billion have preventable or treatable conditions [WHO 2019]. In the US, about 12 million people 40 years and over have vision impairment, including 1 million who are blind, and nearly 3% of children younger than 18 years are blind or visually impaired [CDC 2020b]. People who are blind may read *Braille*, a pattern of raised dots felt with fingers, or use screen readers to audibly say what is on the screen. At the other end of the spectrum is *color blindness*, which is a decreased ability to distinguish between different colors. For example, the most common is red–green color blindness, which occurs in 8% of men and 0.5% of women of Northern European ancestry.

17.3.3 Deafness and Hearing Loss

Hearing loss is a decreased ability to hear, and total *deafness* is defined as the inability to hear sufficiently well so that spoken language can be understood, even

with amplification. Hearing loss may be caused by injury to the ear or brain or by various diseases. People with hearing loss may communicate with other people through sign language instead of speaking, but generally can use most interaction techniques in this book without modification except for sound and speech—for example, in videos, which must be subtitled or captioned.

In the US, more than 35 million people (13%) report some degree of hearing loss, and approximately 1 million people (0.38% of the population) over 5 years of age are "functionally deaf," more than half of whom are over 65 years of age [Reilly and Qi 2011].

17.4 Interaction Techniques Related to Disabilities

Given the wide range of disabilities, there is a corresponding wide range of adaptations and devices that try to help. Some of these are physical devices that help mimic or replace pointing devices and keyboards, and others are software techniques to change and adapt the on-screen widgets.

17.4.1 Adaptive Input Devices

Some people with physical disabilities are able to replace a mouse with a trackball or joystick or other off-the-shelf pointing device. For example, in one of our studies [Myers et al. 2002b], a 27-year-old with MD had lost the ability to use a regular mouse or keyboard, but he was able to use a trackball for pointing and two pencils to press keyboard keys (see Figure 17.3). We also found that using a stylus on a Palm screen was feasible for people with MD since they had good fine motor control but little strength, and we provided a remote control app on the Palm where these users could control the computer's mouse and keyboard using a Palm device [Myers et al. 2002b]. Others with more severe limitations may use a "head mouse" as a pointing device. One common form uses an IR transmitter/receiver mounted on the monitor, which reflects off of a small reflective dot sticker attached to the user's forehead [Hansen et al. 2004], but there are a variety of other types of head or gaze trackers. Gaze tracking or eye tracking has also been used to control a pointer for people with little other muscle control. Other people may use *mouse keys*, where the mouse pointer is moved using a set of four or eight direction keys, which was originally built into the X window system to allow the mouse to be simulated by users who didn't have or didn't want to use a mouse but was also found to be useful for accessibility.

For the mouse *buttons*, users may have a large physical button that they can press using whatever body parts they can use, such as their arm, hand, head, or chin. Another option is a *sip-and-puff device*, where the user can exhale (blow) into

Figure 17.3 Dan was a 27-year-old with muscular dystrophy in 2001 who typed using two pencils and used a trackball (on the pink wrist pad) for pointing. Source: Image taken by author, originally published in Myers et al. [2002b].

a tube for one kind of action, like pressing a button, and suck in (inhale) for a different action (such as the right mouse button). Typically, the tubes are mounted flexibly so the user can still move the pointer with their head while holding the pressure. Sip-and-puff devices actually predate personal computers, having been available since the 1960s (see Figure 17.4) for use with a typewriter. Another option is to register a press if the user keeps the cursor pointing at a location for a specific duration of time (dwell time), but this obviously has problems with false positives [Hansen et al. 2004].

17.4.2 Selecting Items by Scanning

For people who cannot control a mouse pointer to select a desired item, either due to lack of sufficient motion control or else due to a visual impairment so they can't see where on-screen items are, a *scanning* approach can be used. With scanning, the computer moves the selection from item to item, and the user can use a single button or keystroke to tell the movement to stop and thereby select the desired location. For people with visual impairments, the computer may read out the items audibly whereas for sighted people the items can be graphically highlighted. This

Figure 17.4 Early 1960s picture of user operating an electric communication device in the form of a sip-and-puff typewriter controller named the Patient Operated Selector Mechanism (POSM or POSSUM), which was developed in the United Kingdom. Source: By Possum Ltd. Released into the public domain by Possum company, https://commons.wikimedia.org/w/index.php?curid=16294947.

approach is frequently used with on-screen keyboards (see Section 8.3.6), but it also can be used to select among a collection of icons or other menu options.

The speed of the movement of the selection (or reading out the options audibly) can be adjusted to match the user's abilities. Another speedup is to narrow down the selection in groups, thereby making the time to reach the desired item be a log_n search instead of linear. For example, in an onscreen keyboard, the selection can first move line-by-line or column-by-column, then to the individual key, or else half of the items can be selected, then half of the selected part, and so on, to get a log_2 search time. Still, the measured input rate for a *scanning keyboard* may be as low as 1 wpm for people with severe disabilities [Damper 1984].

17.4.3 Text Entry

In addition to the on-screen keyboard approach of the previous section, many other approaches for assistive text entry have been explored. Users with visual impairments can often use conventional computer keyboards by placing their fingers in

the correct places by feel. However, they will need some indication of whether various keyboard modes are on or off, such as CapsLock, ScrollLock, NumLock, or Insert mode, since these are often indicated by lights on the keyboard. For some users who only have one hand or even one finger available, it is impossible to hit modifier keys like Shift and Ctrl simultaneously with characters, so researchers at the Trace Center developed *Sticky Keys*, an interaction technique where the user first hits the modifier key, and it stays active until the next key is hit [Vanderheiden 2008] (which is how onscreen keyboards work for the Shift key). For some users with more involved physical disabilities, sometimes speech-to-text can be used (Section 18.3), where the user dictates the desired text, but many physical disabilities also affect speech so the speech recognition accuracy may be lower. As mentioned above, we found that certain users with physical disabilities were able to tap on the Palm onscreen keyboard for text entry, and as explained in Section 8.3.2, we created a new text entry method called EdgeWrite involving single-stroke gestures that can be made reliably by people with a variety of disabilities (Figure 8.9).

17.4.4 Augmentative and Alternative Communication Devices

People who lose the ability to talk, due to a disorder listed above or an operation or medical condition, may use an augmentative and alternative communication (AAC) device to communicate with others. A simple form of AAC device is simply a piece of cardboard with multiple pictures to which a user may point. Typically, each picture represents a whole concept, like "I am hungry" or "I need to go to the bathroom," but they can also be individual words or letters that users can string together to make sentences.

Since the 1960s, there have been computer-assisted devices that communicate the selected items. For example, the Prentke Romich Company produced its first communication device using a Teletype machine in 1969, a portable device in 1979, and the first device to use synthesized speech capabilities in 1982 [Romich 2021]. These devices often have multiple panels, some with on-screen keyboards but others with arrays of pictures for faster sentence authoring. Still, the rate of composing speech is quite slow, so AAC users will often pre-author sentences in advance when possible. As shown in Figure 17.1, Stephen Hawking used an AAC device for communicating after he lost the ability to talk.

17.4.5 Brain–Computer Interfaces

Brain–computer interfaces (also called *brain interfaces*) are when signals in the brain are measured to directly control a computer function. Various technologies have been used to measure the brain's signals, with the most popular being an *electroencephalograph* (*EEG*) that measures the brain's electrical signals from the outside

(a) (b)

Figure 17.5 (a) Red cap contains EEG electrodes. (b) Monkey at right has implanted electrodes in the brain and has learned to control the robotic arm. Sources: (a) Fig. 1 of Peck et al. [2010]. (b) BBC, Courtesy of the University of Pittsburgh Medical Center, Fair use, https://en.wikipedia.org/w/index.php?curid=23124084, from Baum [2008].

using electrodes arranged around the user's head, often on a tight-fitting cap (see Figure 17.5(a)) [Peck et al. 2010]. EEGs and similar devices obviously have medical and study applications but as an input device have many limitations, such as being very sensitive to muscle movements and having low spatial resolution. However, researchers have shown that people with physical disabilities can learn how to control their brain to produce detectable signals for selecting objects in a scanning keyboard or even controlling a wheelchair's movement [Peck et al. 2010].

Microelectrodes can also be implanted inside the gray matter of the brain using neurosurgery, which can get cleaner signals and can also provide feedback to the brain, but these are obviously much more invasive and come with significant risks. Whereas this sounds like science fiction, cochlear implants are now routinely performed for people who cannot hear to provide electric signals that directly stimulate the auditory nerve. Researchers have also shown that monkeys with implanted electrodes can learn to control a prosthetic arm (see Figure 17.5(b)) [Baum 2008].

17.5 Screen Readers

A *screen reader* is software that audibly reads out loud the contents of the screen, primarily to help people who are visually impaired know what is on the screen.

As explained in Section 2.3, early terminals supported only character display and input, and so it was easy for tools to read the contents. An article from 1982 [Stoffel 1982] lists three commercial "talking terminals" and discusses the speech output quality and issues with the user interfaces for controlling what is being read, such as pausing, backing up by different amounts to re-read parts, and switching to spelling if the word is mispronounced or unintelligible (like for proper names or foreign words).

Displays for early personal computers, such as for MS-DOS (Section 2.4.8), were also easily accessible for screen readers. One of the most popular screen readers for DOS was Freedom Scientific's *JAWS*, which stands for Job Access with Speech, and was first released in 1989. However, with the advent of GUI screens, containing many icons, other images, and 2D layout, a more sophisticated approach was needed so that visually impaired people could understand the content of the screens. JAWS was modified to work with Windows in 1995 as GUIs started to become dominant. It provided a large number of keyboard commands to navigate around the many windows and their widgets. Besides reading out the contents of the screen, it also allowed users to operate the widgets, such as pushing buttons and scrolling using only the keyboard. Today, the Windows, Macintosh, iOS (Apple iPhone, iPad, etc.), and Android operating systems all have screen reading built in.

As the web became more popular, screen readers were adapted to help the visually impaired be able to navigate. The increase in difficulty paralleled what happened with desktop interfaces—with the original simple web pages supported by HTML (see Section 2.4.13), screen reading was easier, but as Flash, divs, and more complex formatting became popular, more sophisticated commands for navigating became necessary. The IBM Home Page Reader (HPR) was created by IBM researcher Chieko Asakawa originally for Netscape since the generic screen readers of the time did not support the web very well, especially 2D material like tables, and was released by IBM as a product in 1997 [Asakawa and Itoh 1998]. A special mode was created for JAWS to make it work better for web browsers, adding commands to jump from header to header, around a table, through all the links, and so on.

One interesting control for screen readers is the *speed* at which the content is read. People who are experienced with screen readers often can understand the speech at rates that sound like gibberish to untrained ears. Rapid normal speech is about 6 syllables per second, or about 216 wpm, whereas blind users can understand screen readers at speeds of up to 25 syllables per second, which is an amazing 900 wpm [Fields 2010]! Compare this with Table 8.1 that shows the normal speech output speed at 100–150 wpm, the fastest typing speed at 200 wpm, and normal reading is around 300 wpm. Functional MRI brain scans of people who were blind

from birth revealed that the part of the cerebral cortex that normally responds to vision was responding to speech [Fields 2010].

17.6 Adaptations Built into PC and Macintosh

As discussed above, a wide range of adaptations are needed for interaction techniques to assist people with a wide range of disabilities. Although some of these can be addressed by third-party applications like screen readers, most require changes to the underlying operating systems. Therefore, researchers and companies worked together to add such features. For example, the Trace research center collaborated with Apple, IBM, Microsoft, and Sun Microsystems throughout the 1980s and 1990s to add disability access features to their operating systems. The Sticky Keys interaction technique for keyboard modifiers was adopted by Apple in 1988 for System 6 as part of the Easy Access extension, which also included mouse keys functionality, but accessibility was not built-in as part of Microsoft Windows until 1995 [Vanderheiden 2008].

Modern operating systems provide these and many other modes that can be enabled. For example, Figure 17.6 shows the Windows 10 "Ease of Access" control panel and the Magnifier control. "Start Narrator" is the built-in screen reader, and the Magnifier supports multiple ways to zoom into parts of the screen. The other options control the way other interactions work. The options for Macintosh systems are similar.

17.7 Adaptations Built into Smartphones and Tablets

Many of the adaptations for regular computers have been brought over to smartphones and tablets, but the interaction techniques may be different (Figure 17.7). For example, mobile devices have a screen reader, but the lack of a keyboard means that people who are visually impaired need a different way to control it. One approach is that the user can feel the edge of the device, and then move their finger slowly down or across, and the screen reader will read out what is in that vicinity, so the user can stop when at the right place. I find the adaptation that magnifies the screen of my iPhone to be very useful—after being turned on, tapping two times with three fingers toggles zooming in or out, and the same three fingers can be used to scroll around while zoomed in, to make the screen easier to read. Alternatively, the equivalent of Sticky Keys is available, so the user never needs to use multiple fingers at the same time (e.g., for zooming into a map). A standard protocol for communicating with external buttons through Bluetooth is available. Furthermore, the screen-side camera on the phone can even be used as a head tracker so a user can control an on-screen cursor via head movements.

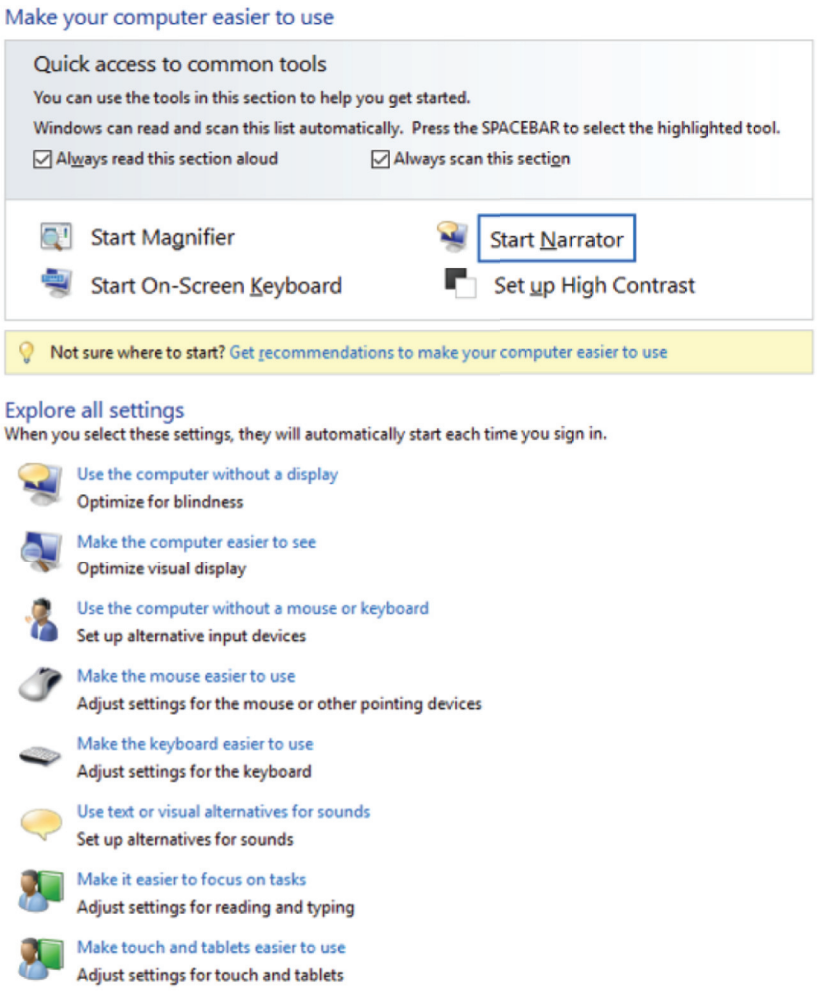


Figure 17.6 Microsoft Windows 10 “Ease of Access” Control Panel and Magnifier control. Source: Screenshot on February 12, 2021, used with permission from Microsoft.

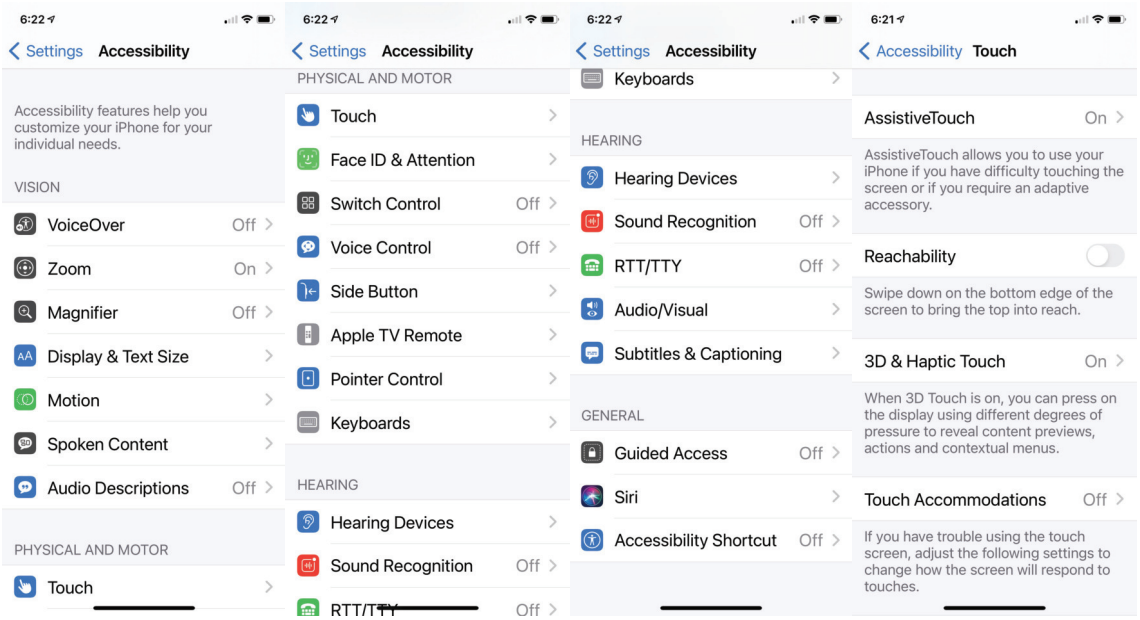


Figure 17.7 Some of the many screens of controls for the iPhone's accessibility features in version 14.4. Source: iOS screenshots on February 12, 2021.

17.8 Web, App, and Consumer Electronics Accessibility

Web accessibility is the practice of reducing or removing the barriers for all people to successfully read and interact with websites and their content. Similarly for apps on mobile devices. While the discussions of the previous sections still apply to web pages and apps, there are a large number of additional requirements and recommendations that apply. An obvious example is that all pictures should have an "alt text" that can be read by screen readers to say what the image is. This is especially important for images that are used as functional icons—a blind person would not know that a picture to be clicked on is a forward arrow if the screen reader can only tell it is an image with the URL of "`picture12.jpg`." Other requirements include using meaningful tags, such as headings, rather than formatting that achieves the same look, such as bold and bigger font, so screen readers can more effectively convey the structure of the page. Another feature for screen readers is an invisible tag in the page source with a link to the start of the main content, which may be obvious for sighted users but confusing for a screen reader that would otherwise need to read through the page navigation, ads, and other content at the top of the page. Users with auditory impairments may need captions on all videos or other spoken

content. Users with physical or mental impairments may take longer to interact with pages and may benefit from longer timeouts.

The specifications for HTML and its related languages (like CSS and JavaScript) have been updated to include these kinds of features for accessibility, and using these on web pages has become a goal and requirement, as mentioned in Section 17.1. For example, the Web Accessibility Initiative (WAI) of the World Wide Web Consortium (W3C) developed multiple versions of their recommendations for making Web content more accessible, starting with the Web Content Accessibility Guidelines (WCAG) 1.0 in 1999. WCAG 2.0 came out in 2008 and became ISO standard ISO/IEC 40500:2012 in 2012. Various laws, including the Rehabilitation Act of 1973 in the US and Directive 2016/2102 in the European Union, have been used to require websites to meet at least some of these guidelines.

One interesting aspect of web accessibility is the *Captcha*, which are elements of a web page designed to make it difficult for the page to be used by a program (often called a "bot," short for robot) instead of by a human—for example, for buying concert tickets or registering for a new email address. These were invented in 2003 by CMU Ph.D. student Luis von Ahn and his advisors and originally were pictures with distorted characters where the user was required to type in characters. The name is an acronym for "Completely Automated Public Turing test[1] to tell Computers and Humans Apart" [von Ahn et al. 2004]. It would totally negate the purpose if these images had their contents in a machine-readable tag, so obviously, these are completely impossible to use for people who need a screen reader. As these were more widely deployed, an alternative where blind users could type in what was said in a distorted audio file was provided, but in one study, 89 blind web users were only able to achieve a 43% success rate in solving 10 popular audio Captchas [Bigham and Cavender 2009]. The researchers were able to improve the success rate by 59% just by changing the way playback and answering was controlled [Bigham and Cavender 2009].

Similar requirements and guidelines apply to applications built for desktops and mobile devices, such as smartphones, even though they may be built using toolkits instead of HTML. Many of the popular platforms have guidelines for accessibility, such as the iOS and macOS "Human Interface Guidelines" sections on accessibility[2] that start with an example of image descriptions. The original Apple accessibility guidelines were developed in collaboration with the Trace Center [Trace 2021]. Similarly, Android's "Make apps more accessible"

1. The "Turing Test" concept is explained in Section 18.3.

2. https://developer.apple.com/design/human-interface-guidelines/accessibility/overview/content/.

guidelines in the standard documentation[3] includes a requirement for a textual `contentDescription` for elements to be read by screen readers. People with disabilities may also benefit from using speech interfaces to these apps. However, in our research [Li et al. 2017], we found many examples of Android apps, some even from large companies, that did not follow the guidelines, such as elements with no descriptions and controls that went around the standard Android mechanisms (e.g., using a tiny web browser window for a single button) to achieve a certain effect, but as a consequence, made them completely inaccessible to anyone needing any kind of augmentation.

Consumer electronics and appliances are particularly difficult for people with disabilities to use. Many have touchscreens or flat digital touch pads with no tactile distinctions, and even ones with physical buttons may have all output shown only on a screen. People with physical disabilities who cannot push hard or accurately or people with visual impairments who cannot see the buttons or the feedback are prevented from using these appliances at all, and virtually none have accessibility features. As mitigations, the Trace Center has shown how important devices such as ATMs, public phones, and voting machines can be accessible, which has influenced requirements for them [Trace 2021]. My group had a research project that investigated moving the control of appliances onto the user's smartphone [Nichols et al. 2002], which could then use the smartphone's accessibility features. A research project by others at CMU showed how blind users could have 3D printed Braille stickers attached to the buttons of appliances so they could identify what each did by feel [Guo et al. 2017].

17.9 Summary of Consideration for Interaction Techniques for People with Disabilities

No matter what platform you are designing your applications or interaction techniques for, you should pay attention to making them accessible for people with a wide variety of abilities. This is getting to be increasingly important as the number of older people and people with disabilities using computers increases. As mentioned above, being accessible is required by law for many products, and lawsuits have been won against companies that do not comply. Currently, the best way to know what to do is to understand and follow the guidelines for the target platforms and to correctly use the appropriate features of the underlying implementation tools. However, this will take planning and effort for the design and implementation team.

3. https://developer.android.com/guide/topics/ui/accessibility/apps.

17.10 Chapter Conclusions

Although making interfaces and interaction techniques more accessible is clearly the right thing to do, it does come with costs. Besides the extra effort to understand and comply with long lists of accessibility requirements, there is also the impact on creativity—if the designer is interested in exploring or designing a new kind of interaction, it may be less clear how it will impact people with disabilities or how to adapt it. Cutting-edge interactions, such as for VR/AR (Chapter 12) or intelligent interfaces (Chapter 18), have very limited research or development on how to make them accessible. And there is very little work or guidelines on how to make interactions more accessible to people who have *cognitive* disabilities. On the other hand, there continues to be active research on all of these topics, including interesting research on very different interfaces, such as direct brain sensing, that may be feasible in the future.

CHAPTER 18

Intelligent Interfaces: Artificial Intelligence in Interaction Techniques, Speech Recognition, and Conversational Agents

18.1 Introduction

The way that people interact with computers in most science fiction stories is by talking to an intelligent agent. Recently, intelligent agents such as Siri and Alexa have begun to be commonly used. Although this chapter seems like it covers a lot of different topics in which there is an enormous amount of research and development, there is only a little that is relevant to interaction techniques. For example, this chapter does not try to cover speech recognition technology or natural language processing beyond the basic ways it is used as interaction techniques. There are many issues of interest to HCI researchers and practitioners that are also out-of-scope for this book, such as artificial intelligence (AI)'s impact on society and the fairness and bias of AI.

The promise of intelligent interfaces has been well-covered—they can provide a natural way for people to interact with machines that requires little learning, can make users more efficient, would open up more capabilities to more users, and be accessible to users with a much broader range of abilities. Another potential advantage is that intelligent interfaces, especially using natural languages (like English or Chinese), could provide a higher level of abstraction than direct manipulation interfaces since people could say their higher-level goals (e.g., "book a flight to New York City") rather than requiring a large number of clicks and decisions that would

be required by a direct manipulation interface. Of course, there are also many challenges and risks with intelligent interfaces, including lack of affordances, unclear scope of what can be said, loss of feedback and control, incorrect actions with little or no way to review or correct, loss of privacy, and so on (see Section 18.8).

The idea of making computers more "intelligent" has been around since the 1950s. Many of the early systems were trying to *replace* people, such as computers autonomously playing checkers, controlling robots, or proving theorems. However, having people interact with computers using natural language has been investigated at least since the ELIZA program in 1964 [Weizenbaum 1976]. Whereas most AI research is published at AI conferences, there have always been a few publications about intelligent interfaces in HCI venues. Also, there has also been an annual ACM conference devoted to Intelligent User Interfaces (the IUI conference) since about 1995.[1]

Besides using natural language in the form of speech or typing, this chapter covers intelligent interaction techniques such as embodied intelligent agents, intelligent defaults, AI algorithms for spelling and grammar checking, and machine learning algorithms for selecting content for users.

18.2 Definitions

AI is a broad area of computer science that involves making machines behave in an intelligent way. This definition is vague and is subject to controversy. It is sometimes said that AI is anything that we don't yet know how to make computers do reliably, and once we do, it just becomes an algorithm and isn't AI anymore. However, for the purposes of this book, it isn't important to have a crisp definition that can identify whether a technology "counts" as AI or not. Similarly vaguely, I will use *intelligent interface* (or *intelligent user interface*) to cover any kind of user interface that uses AI so that it seems intelligent to the user. One defining feature of all intelligent user interfaces is that (at least from the user's point of view) the system is *guessing*, and therefore may be incorrect, so all intelligent interaction techniques will need to consider the issues of *feedback* about what the system is doing and user *control* to fix any errors in what is guessed.

Speech recognition is the process where a computerized device understands what is being said. More general terms are *natural language processing* (NLP) and *natural language understanding* (NLU), where computers deal with natural languages (like English, French, Chinese, etc.) that may be presented textually or through speech. There are sometimes distinctions made between NLP and NLU as well, with

1. I was supposed to be a keynote speaker at the 25th annual IUI conference in March of 2020, but the conference was canceled due to the COVID-19 pandemic.

NLP being broader, encompassing processing that does not necessarily involve understanding—for example, to just count or analyze parts of speech. NLP also may be used to include *speech generation* or *natural language generation* (NLG), when the computer creates or speaks in natural language.

A *conversational agent* (also called a *virtual assistant*) is a computer system that engages in a two-way dialog with a human. In this chapter, we will often just use the term "agent" to mean a conversational agent. Note that this definition requires that the agent be able to both accept and respond using natural language. When the conversational agent primarily or exclusively uses speech input and output, it is also called a *voice assistant*. Examples of voice assistants are Siri for Apple phones, tablets, and computers and Cortana for Microsoft Windows. *Smart speakers* like Alexa and Google Home, which are physical devices whose main input–output is through speech recognition and sound output, also contain voice assistants. Another class of conversational agents are *chatbots*, which communicate primarily or exclusively through the user typing using a keyboard and the system generating text shown on a screen. The name comes from *chat* interfaces, where multiple people type text to each other, like with text messaging on phones (as in Figure 8.30).

18.3 Speech and Natural Language Interfaces

One of the first natural language interfaces was the ELIZA program created by Joseph Weizenbaum in 1964, which simulated a psychologist asking the user questions that were typed into a console in English [Weizenbaum 1976]. It used fairly simple keyword spotting in the user's answers to come up with the follow-on questions, which many people of the time found compelling and scary, even claiming it solved the *Turing Test*. The Turing Test is an experiment, originally articulated as the "imitation game" by Alan Turing in 1950, where a person is unable to tell the difference between conversations in natural language with a computer versus with a person. In fact, it was easy to ask questions to which ELIZA gave answers very different than what a human would; today there are systems that perform quite well on a Turing test but only in very limited domains—general purpose conversational interfaces are still a long way off. While ELIZA was created as a playful experiment [Weizenbaum 1976], other speech and natural language interfaces of the era were being researched to perform more serious tasks, such as Terry Winograd's 1972 dissertation on the SHRDLU system for giving commands in a simulated blocks world [Winograd 1972].

Skipping ahead through about 40 years of slow but steady progress in speech and natural language processing, including a number of paradigm shifts in the underlying technologies being used, starting in the 2010s we see a variety of speech

interfaces finally becoming good enough for use in general-purpose consumer devices, like Siri for the iPhone.

But what counts as "interaction techniques" for speech and natural language interfaces? To match what was discussed in the previous chapters, I will include: what can be said, how the user provides input, how the computer provides feedback about what was input, how the user can verify and correct that feedback, the actions that result, and to what extent they can be canceled and undone.

18.3.1 Kinds of Dialogs and What Can Be Said

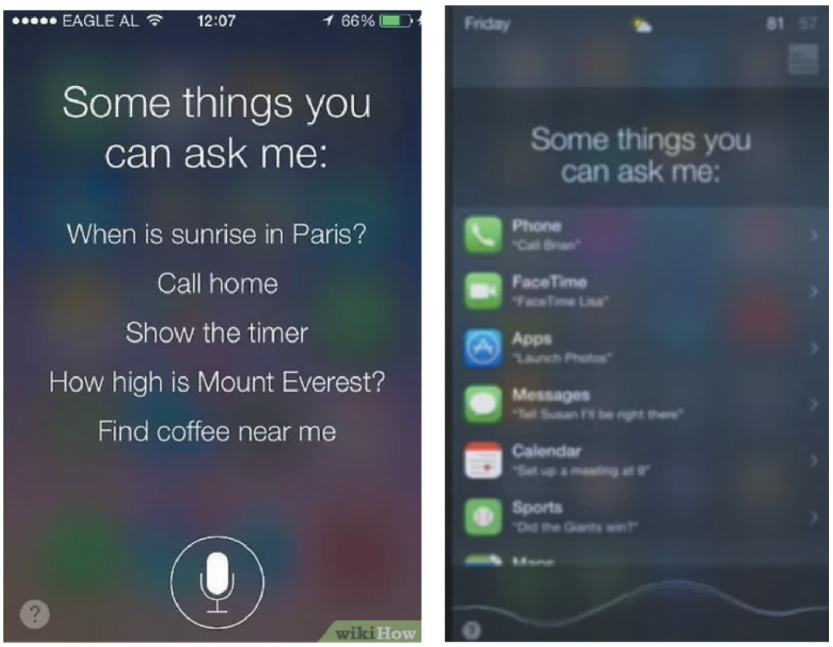
The key problem with speech and natural language dialog systems is that the user does not know what can be talked about, and how it needs to be phrased. There is little way to provide affordances to manage the user's expectations or to communicate the agent's capabilities [Lieberman et al. 2004]. This is especially difficult for general purpose speech interfaces such as iPhone Siri or Amazon Alexa, where the user could potentially talk about almost anything. Special-purpose systems, such as a chatbot for tech support for a company, have it a little easier in that users will know that they should talk about what that company provides, but there still is no guidance about *how* requests can be phrased.

Compounding this problem is the lack of good error messages and feedback (see also Section 18.3.3)—when the system doesn't understand the user, most systems provide only generic feedback ("I don't understand"), providing no guidance as to whether the system could have handled the request if it was phrased differently, versus the request being out-of-scope no matter how phrased.

One approach that a number of systems have used is to provide prompts on the device about the kinds of things the user can say (see Figure 18.1). Off-device help is also usually available, such as training manuals and demos about what can be said. However, these are obviously very limited approaches.

Another dimension to this issue is whether the system supports multi-turn dialogs versus just single commands. Most of today's systems are similar to the original *Put-That-There* system from the 1970s (see Section 18.4) supporting mostly simple single-utterance commands or questions, with the possible addition of filling in missing parameters that are necessary [Bolt 1980]. More complex dialogs, which might include references to previous actions (including pronouns), explanations, clarifications, and so on, are rarely supported even by today's best systems.

As a consequence, most people do not use the full potential of dialog systems and just limit themselves to simple commands ("what is the weather today?") that they know will work, or they just use it for simple transcriptions of short emails or text messages.



(a) (b)

Figure 18.1 (a) Version of the Apple Siri screen from around 2013 showing prompts. (b) 2015 version adds prompts for various apps. Sources: (a) Provided by wikiHow at https://www.wikihow.com/Get-Siri-on-Any-iOS-Device; CC-BY. (b) iOS Screenshot.

18.3.2 How Users Provide Input

The first choice is whether the user types or talks. The key distinction for speech interfaces is whether the user needs to indicate that they are going to start talking, which is sometimes called *push-to-talk*. For example, you might tap or hold down the on-screen button in Figure 18.1 or the home or power button on the iPhone to indicate it should start listening. The iPhone currently has multiple places to click to commence different kinds of listening—in Figure 18.2(a), the user can start a global command by holding in the power button on the side, start an audio recording to send the audio file as a message using the icon in the message field, or start dictating to have the speech converted into text using the icon below the keyboard. Automobiles usually have a button on the steering wheel to indicate the speech system should start listening. As discussed in Section 12.4.7, the Google Pixel 2 phone from 2017 added sensors on the sides that are primarily to support *squeeze to talk* [Quinn et al. 2019]—see also Chapter 35. Researchers have looked at triggering having a mobile device start listening for speech when the device is tilted into a particular orientation [Hinckley et al. 2000], and this was available in an early

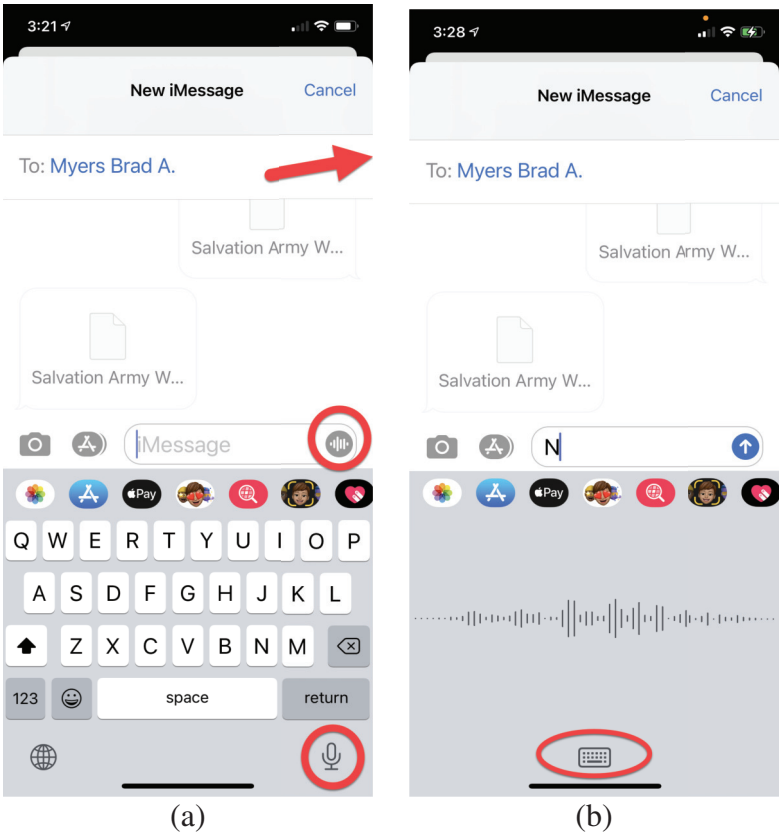


Figure 18.2 (a) Three places to initiate listening on the current iPhone Messages app—the power button on the side for commands, the icon in text field to record an audio message, and the icon on the keyboard to start transcriptions for dictating the message body. (b) While dictating, the user can be silent or press the keyboard icon to stop. Source: Screenshots of iOS 14.4.

version of Siri, but it has been removed due to too many false positives—it starts listening when the user didn't want it to.

The primary alternative to push-to-talk is to have the device always listening, typically for a special *trigger phrase* to signal that it should start interpreting what it hears as input. For example, "Hey Siri" for the iPhone or "Hey Google" for Android. Smart speakers like Alexa or Google Home must have this technique as an option since they are designed to be used from across the room and don't always even have a button. Often the device will confirm that it heard the trigger phrase and is listening by making a tone or speaking a phrase (like "I'm listening"). One downside for always listening is privacy—the microphone of the device must always be

on, and users don't know what the device is processing locally versus sending off to remote servers.[2] The other key disadvantage is false positives, where the device starts processing ambient speech when the user did not intend it to, potentially resulting in inappropriate or incorrect actions.

To end the input, textual (typing) natural language interfaces like chatbots generally use the keyboard Enter key or an on-screen send button. When a speech system uses push-to-talk that requires holding down the button while speaking, then the signal to end the speech is releasing that button. When a button is pressed and released (clicked or tapped) to start, then typically the speech is ended by the user being silent, or there might be another button to tap to end or abort the listening, as in Figure 18.2(b). Systems that are always listening usually stop listening by detecting silence.

18.3.3 Feedback, Verifying, and Correcting

As mentioned above, a key problem with speech and natural language interfaces is the user feedback and control—how the user knows what the system understood the input to be, what the system will do in response, and how the user can stop or correct it if the interpretation is erroneous. There are multiple phases of this process.

The first phase for speech interfaces is to convert the speech into textual natural language. Often this is the easiest to provide feedback for—if there is a screen, the system can simply print out the text of what it heard, or it might repeat back verbally the utterance and ask something like: "Is that correct?". Corrections at this phase are simple—the user can edit the displayed text or say "no" and re-state the utterance. One study identified that when users suspect that the system has misheard the utterance, they might apply prosodic changes (adjusting the rhythm or cadence of speech), use over-articulation (exaggerating sounds) or increased volume, or simply repeat the original utterance [Beneteau et al. 2019].

All the next phases are much more difficult to provide good UIs for, however, and this applies to both typing and speaking natural language systems. Different recognition algorithms work in different ways, but they all must have a part that (1) understands the specific words from the user and their order (the vocabulary and syntax) when the system is *parsing* the input, and a part that (2) understands the *intent* or *semantics* of what the utterance *means*. Many systems, both historical and current, do not provide adequate feedback about what the problem is when either of these fail, and the systems just give generic feedback like "I do not understand"

2. Most of today's devices only listen locally for the trigger phrase and do not record any speech or send it to remote servers until the trigger phrase is detected locally.

or "I didn't get that." If the system is a little more confident, it might perform a generic action, like looking up the utterance on the web.

The lack of good feedback is a problem since users would naturally attempt different strategies depending on the source of the error—if the speech was heard incorrectly and produced incorrect text, then the user could try speaking more distinctly, louder, or go somewhere with less background noise. However, if the system has syntax or semantic errors in understanding the utterance, those strategies won't help. If the user knew that there was a vocabulary problem, then simpler words or synonyms could be substituted. If it was a parsing problem, then maybe a simple sentence structure with shorter utterances would help. If it was a semantic problem, then the user might understand that the desired action is out-of-scope and cannot be handled by the system. In a few cases, the programmer might have supplied a task-specific clarification response with an error handling conversation flow, but these are rare. In all other cases, users must *guess* which of these repair strategies to try, but today, whatever the user tries rarely works, so users generally give up after the first error and revert to simpler requests.

There has been some research on this *breakdown repair* in natural language interfaces. One approach is *mutual disambiguation*, where inputs from multiple modalities complement each other so that what is clear in one modality can be used to help resolve breakdowns in another [Oviatt 1999]. For example, studies I participated in from the late 1990s showed that repair success rates would increase from 58% with just respeaking up to 85% if the user switched to handwriting with a pen [Suhm et al. 1996]. The Quickset system from around the same time used a specialized natural language understanding system that took into account input from two modalities: both what the user said and what the user was pointing to or gesturing at with a pen, which actually reduced the number of errors in the first place as well as gave the user better error handling capabilities [Oviatt 1999]. Our recent Sugilite multi-modal research system [Li et al. 2017], which supports speech and programming by example (PBE) to teach new tasks using Android phone apps, had a whole subsystem called Sovite for error handling, where users could point to the GUIs of phone apps to correct errors in the natural language understanding [Li et al. 2020]. Sovite's design highlights the importance of *consistency* between how users discover the errors and how they fix them (see Figure 18.3). Once the users discover an error (the wrong phone app was selected, the wrong screen of that app is proposed, the wrong elements on that screen are highlighted, wrong values from the speech are proposed to fill in those elements, etc.), the ways to fix them are rather intuitive and obvious (e.g., saying the correct app, pointing to the correct screen, dragging the highlights to the correct places, and selecting the right parts of the initial utterances to be used as the values). This was noticed and praised by many participants in our user study [Li et al. 2020].

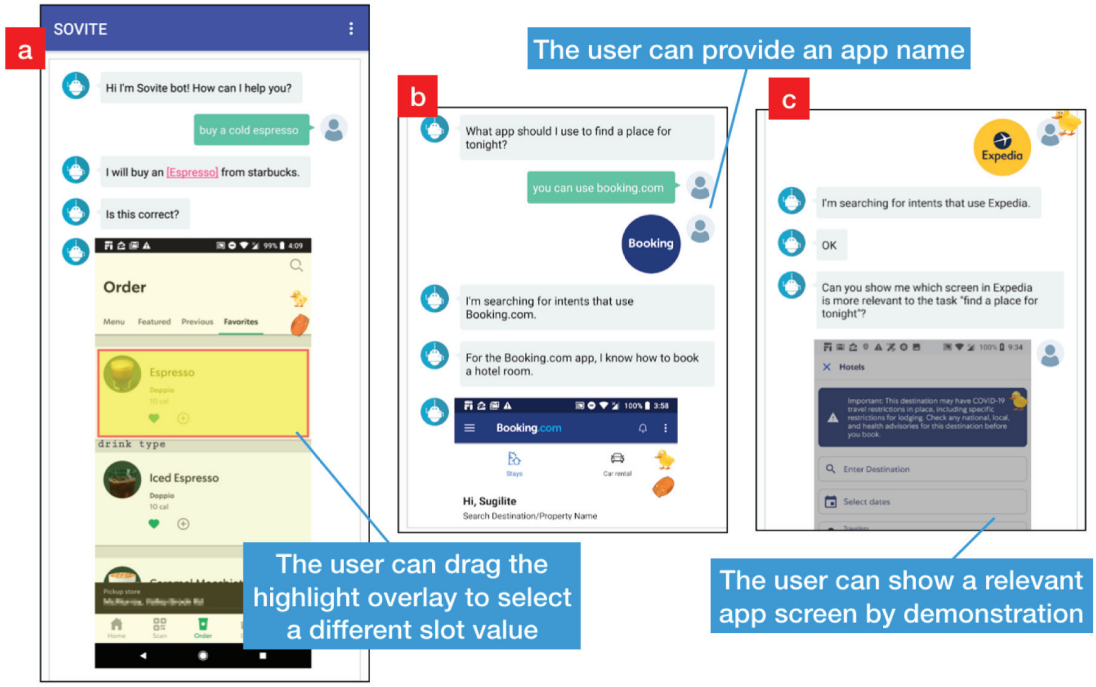


Figure 18.3 The interface of Sovite: (a) Sovite shows an app GUI screenshot to communicate its state of understanding. The yellow highlight overlay specifies the task slot value. The user can drag the overlay to fix slot value errors. (b) To fix intent detection errors, the user can refer to an app that represents their desired task. Sovite will match the utterance to an app on the phone (with its icon shown) and look for intents that use or are relevant to this app. (c) If the intent is still ambiguous after referring to an app, the user can show a specific app screen relevant to the desired task. Source: Fig. 1 of Li et al. [2020].

18.3.4 System Actions

If the understanding step is correct, the next phase is to determine what the system should do in response. If more information is needed, then the system might prompt the user to supply it, either through natural language or a GUI (if available). For example, if I say, "Call Brad" to my iPhone (in iOS v14.4), it says, "Which one" and shows a list on the screen of people named "Brad" in my contact list, so I can either tap on one or say something to disambiguate.

If the system takes some action as a result of the utterance, it may ask for confirmation or just go ahead and do it. Hopefully, this will depend on the seriousness of the action and whether it is easily undoable. For example, if I say, "create an alarm at 4:00pm," my iPhone will just do it, but if I say, "send a text message to my wife that I'll be late," it shows me the draft message before sending it. The alarm is

easily undone by just saying "Delete it." (However, if I say "Undo" instead, it looks up the word "Undo" in Wikipedia!)

18.4 Multi-Modal Interaction Techniques

Multi-modal user interfaces combine any two or more modes of input in a coordinated manner, but this term is almost always used for UIs that combine speech with direct manipulation or gestures. One of the most famous early examples was the *Put-That-There* system mentioned above (Sections 12.4.2 and 18.3.3), which combined speech with pointing [Bolt 1980]. This system also highlighted the importance of coordinating the *time* between the modalities since the system needed to understand which location pointed at went with "that" versus "there" (see Figure 18.4).

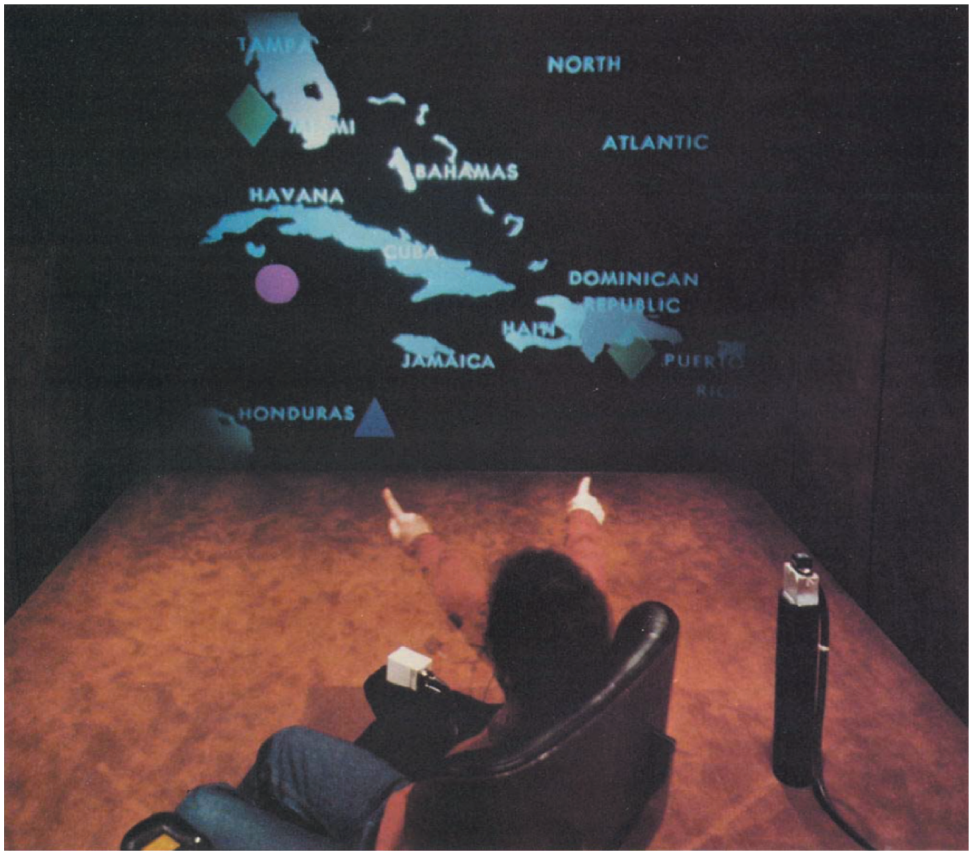


Figure 18.4 Put-That-There combined speech with pointing gestures as an early multi-modal interface. Source: Fig. 2 of Bolt [1980].

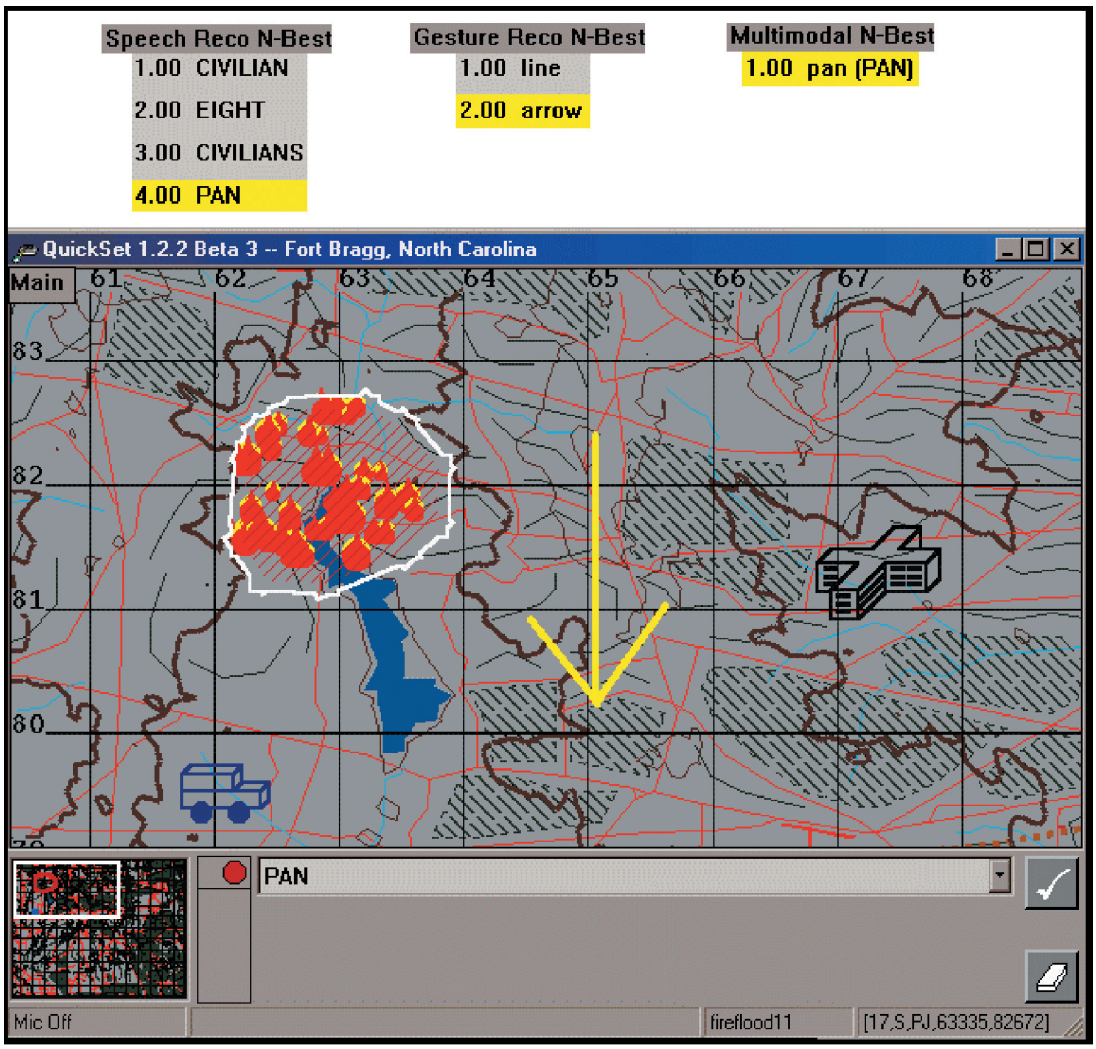


Figure 18.5 In Quickset, the user could speak (here: "Pan") and gesture (the yellow arrow) and the system would combine them to recognize what to do (here, pan down). Source: Fig. 18.3 of [Oviatt 2012].

Another important early system was Quickset (Figure 18.5), which showed that combining speech and pen gestures was more reliable than either alone [Oviatt 2000]. People would learn what modality (gestures or speech) worked best for which kinds of utterances and, importantly, would switch modalities if one had an error [Oviatt 1999]. As mentioned in Section 18.3.3, we leveraged multi-modal input (speech along with PBE and pointing to GUI elements) in Sugilite [Li et al. 2017].

The recent DoThisHere [Yang et al. 2020] system uses a similar multi-modal interface for cross-app data query and transfer between multiple mobile apps.

18.5 Embodied Intelligent Agents

An embodied intelligent agent is when there is a character who is the apparent source of the intelligent agent interaction. In science fiction, it is often a robot or an on-screen humanoid or animal. If the agent pretends to be human, it is often said to be *anthropomorphic*. A famous example in the HCI community was the on-screen bow tie–wearing "butler" in Apple's famous *Knowledge Navigator* concept video from 1987 (see Figure 18.6). This was created by Hugh Dubberly and others for a keynote speech that then CEO John Sculley gave, and it showed their concept of a future agent who the user could interact with through speech and gestures on the screen [Sculley 1989]. When I was Video Chair for the ACM SIGCHI conference

Figure 18.6 The intelligent agent in Apple's 1987 concept video is shown with a bow tie in a window. Source: Second picture in Sculley [1989].

in 1992, I organized a "Future Scenarios" video, and the "Knowledge Navigator" was included [Apple 1987].

A well-known (but widely disliked) implemented embodied intelligent agent was Microsoft's *Clippy*, officially named *The Office Assistant* or *Clippit* (see Figure 18.7). To be precise, the Office Assistant could use a variety of graphics as its presented character, but Clippy was the default and hence the popular name for the feature. Clippy was a follow-on to the unsuccessful Microsoft Bob product (Section 11.11.4), and it first appeared in Microsoft Office 1997 and was completely removed by 2007. Clippy would offer to help in various situations that it was pre-programmed to recognize. Unfortunately, Clippy suffered from a number of usability problems:

- *Not helpful*: Probably the key problem was that it was not very helpful. The things it would offer to help with were not difficult, and what Clippy could do did not save much time or effort. It was very much optimized to help with the initial learnability of a few features, which, obviously, most users had already learned.

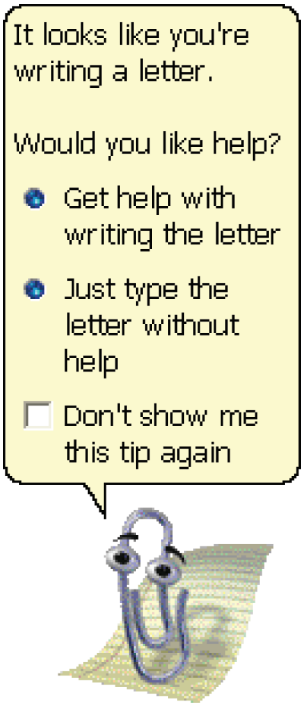


Figure 18.7 Microsoft Clippy. Source: Microsoft, used with permission, https://en.wikipedia.org/w/index.php?curid=1156147.

- *Annoying*: Further, Clippy did not keep track of what the user already knew, and kept popping up with the same questions and unhelpful "help" over and over, unless the user explicitly turned them off, one-by-one (see the checkbox in Figure 18.7).
- *Attention grabbing*: When it was on the screen, Clippy's graphic presentation had a number of usability problems. First, it was constantly moving to give a semblance that it was alive. This was a problem because it usually didn't have anything to contribute to the user, and movement in the periphery of one's visual field is distracting. As explained above in Section 13.2, any movement in such *ambient displays* should be reserved for when it wants to get your attention, and it should not grab your attention when it is idle. Furthermore, when it did want the user's attention, it would animate to pretend to be tapping on the computer glass and make a noise, which grabbed the user's attention even more, even though it was extremely rare that the user would want its help.
- *Creepy*: Another graphics problem was that, although Clippy may have seemed genderless, women who participated in focus groups at the time reportedly told Microsoft that the available Office Assistant characters "were too male and that they were leering at them," but the Microsoft engineers just ignored this feedback [Meyer 2015].

Allegedly (according to Wikipedia), both Bob and Clippy were based on a misunderstanding of Stanford faculty member Cliff Nass's seminal work (e.g., Nass et al. [1994]) on how people naturally treat computers as *social actors* and that interactions can leverage this phenomenon to improve the design of computerized systems. Although Nass was a consultant to Microsoft for a time, he disavowed any connection to the final design of Bob or Clippy, and his recommendations for how to improve it were not followed [Nass and Yen 2010].

As a result of these problems, Clippy has been called "one of the worst user interfaces ever deployed to the mass public" [Meyer 2015] and even broader—"one of the worst software designs in computer history" [Nass and Yen 2010]. Along with other controversy around intelligent agents (see Section 18.8), it is perhaps not surprising that the idea of embodied agents has disappeared from the commercial landscape, except for games and science fiction. Although today's mobile device agents like Siri and Google Assistant, as well as automobile navigation systems, all speak with human-sounding voices, they for the most part avoid being embodied or having much personality.

The one place where embodied agents have continued is with toys and robots, where the embodiment is physical, as discussed in Section 13.3. Some of these are

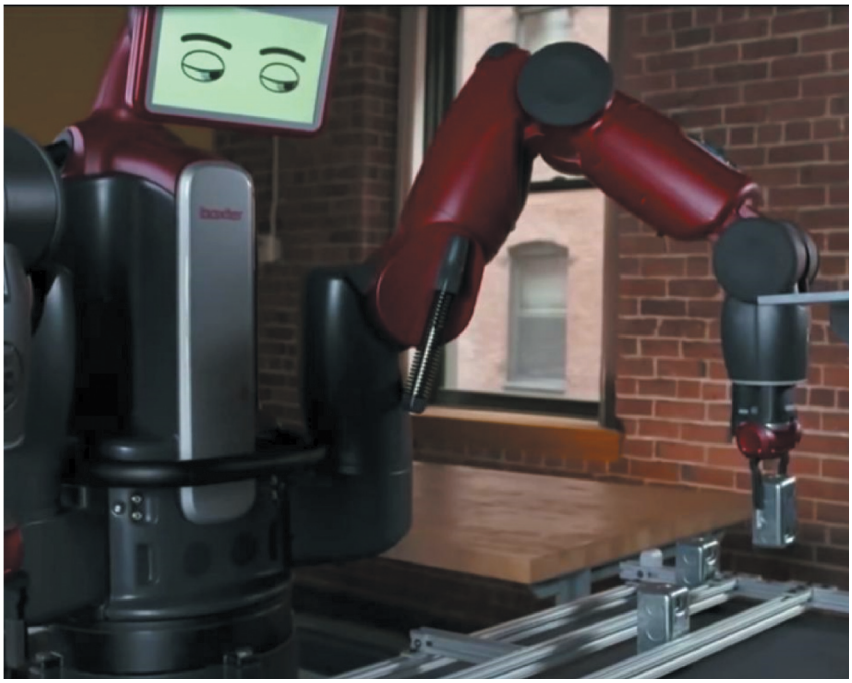


Figure 18.8 The "Baxter" robot from Rethink Robotics from 2015–2018 displayed a face on its screen. Source: By Salammbo31, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=36902398.

humanoid, but typically 1 to 2 feet tall, and another common form is dog shaped, with four legs (see Figure 13.2). As befitting toys, these often try to have personalities and to engage kids in activities that are fun. One recent commercial robot displayed a face on its display screen (see Figure 18.8), but the company failed, mainly due to the robot not performing its intended tasks very well.

18.6 Algorithms and Machine Learning for Content Selection

A completely different kind of intelligence is using complex algorithms, typically trained using *machine learning*, to predict what the user would like to see or do next. The interaction techniques here are presenting the information for the user, allowing the user to pick, and then learning from that selection to inform future presentations or actions. Probably the earliest example of this was research on *recommender systems* (which recommend items to users) using *collaborative filtering* where data from many users is combined to filter which items to recommend, such as which web pages to look at, movies to watch, or songs to listen to Terveen et al. [1998]. Originally, these systems used relatively simple algorithms

to match the user with the selections of other users who were similar to that user in some respects, but the matching and selection algorithms have become more and more complex, taking into account more factors about the user, the content of the items, and what other people have done. The fundamental observation is that there is always too much information for the user to process, so the system must have some way to select what to show or, at a minimum, have a way to sort the material presented. As discussed below (Section 18.8), there are significant concerns about these systems, and there have been situations where the selection has caused harm.

Examples of systems that use this approach today include movie recommenders on sites like Netflix that recommend other movies a user might like based on their past selections, product recommendations on Amazon, selection of which entries to show users on Facebook, Twitter, or Medium, and which ads are shown with all of these. Sometimes, these algorithms are also used to select which search results to show so that different users would see different results even for identical searches done at identical times.

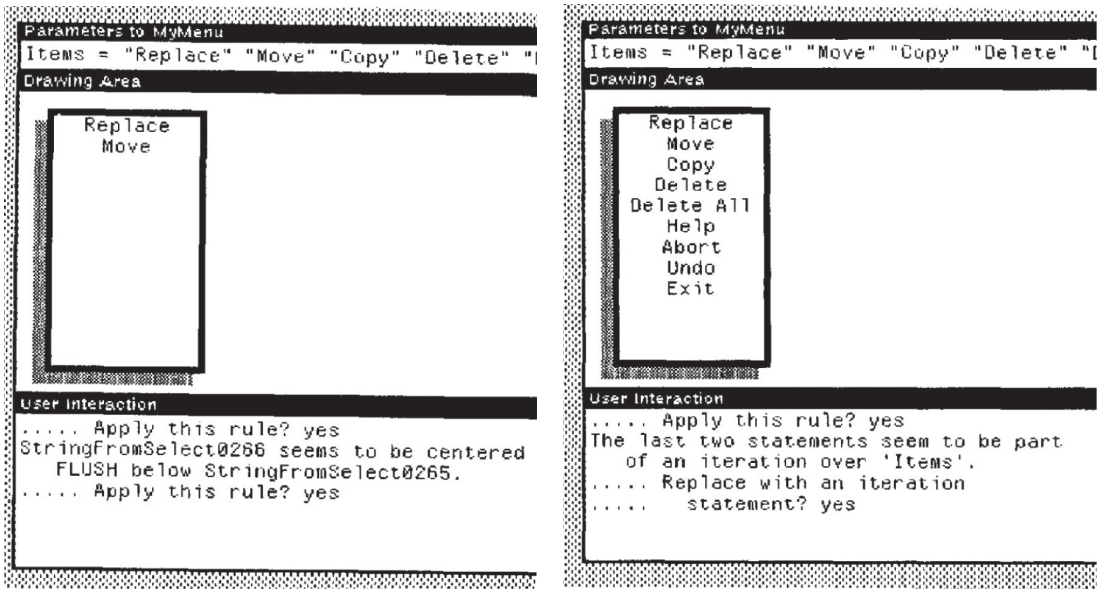
Similar algorithmic approaches have been used to learn other patterns, like to find spam email, credit card fraudulent purchases, which loans to approve, which inmates are more worthy of parole, and so on, but these seem further from "interaction techniques."

18.7 Other "Intelligent" Interfaces

Although conversational agents and machine learning systems may seem like the primary intelligent UI, there have historically been many other ways that AI has been used in interaction techniques, some of which have been covered in previous chapters. For example, some of the earliest intelligent interaction techniques were focused on helping with text entry. Section 8.4.2 discusses DWIM ("Do What I Mean") from 1966 [Teitelman 1966] along with other spell checkers and other ways of correcting what the user typed. Section 8.4.3 discusses autocomplete, when the system guesses what the user might want next, including the way that browsers predict what users might want to search for on the web. And Section 8.4.4 discusses how these techniques were brought to mobile devices.

Apple coined the term *data detector* in 1997 for clicking on different kinds of data in text, such as phone numbers and email addresses, and getting appropriate actions in a menu, such as calling or composing an email message. Apple said these were examples of "programmable intelligent agents" because they "extract[] semantics from everyday documents ... [to] infer high-level goals from simple user actions" [Nardi et al. 1998]. Apple's patent on this technology [Bonura et al. 1996] was one of the ones used in the lawsuits against Samsung.

Another kind of intelligent interaction technique that has already been covered (see Sections 10.9 and 15.13) is PBE, also called *programming by demonstration* (PBD) [Myers 1986a, Cypher et al. 1993, Lieberman 2001], which is where the user records a sequence of operations to be repeated later. The "intelligence" comes in when the system tries to *generalize* from the exact sequence that the user demonstrated so it will work in other situations. For instance, in my Peridot system [Myers and Buxton 1986], if the user performs operations on the first two strings of a list, then it will ask the user if it should use the rest of the strings in the same way. If the user says "yes," then it will replace the two operations with a loop that will always do *all* the items in the list, no matter how many there are (see Figure 18.9; see also Section 20.8 about demonstrating the *behavior*). Various interaction techniques have been explored for the confirmation of the generalization in PBE systems, including question-and-answer like Peridot (Figure 18.9, bottom) and Sovite/Sugilite (Figure 18.3), dialog boxes, menus, showing the proposed result and confirming with the Tab or Enter keys (same as for text completion, Section 8.4), and so on.



(a) (b)

Figure 18.9 (a) When the first two items of the list (shown at the top as "Items = ") are used in the Drawing Area (center), then (b) Peridot proposes to instead use an iteration using all the items, asking the user with a question-and-answer interaction style (bottom). Source: Fig. 3c and d of Myers and Buxton [1986].

Gesture recognition (Chapter 14) might be considered to be a form of intelligent user interface since it involves the system trying to recognize what the user has input. This includes text entry by hand printing or handwriting (Section 14.3.1) and "neatening" of drawings by recognizing what shapes have been drawn (Section 14.3.3).

18.8 Challenges, Risks, and Controversy About Intelligent Interfaces

Science fiction stories have long warned about machine intelligence gone awry, and some prominent HCI people have raised objections to AI in general. In particular, Ben Shneiderman waged a long-running argument against intelligent interfaces, especially intelligent agents, and had a number of debates with agent researcher Pattie Maes in the late 1990s [Shneiderman and Maes 1997]. Shneiderman's main argument was that information visualization and direct manipulation (Section 2.5.1.3) can be more powerful than agents and "gives users the feeling of being in control and therefore they can be responsible for the decisions" [Shneiderman and Maes 1997]. Maes, in contrast, argues for agents because they can be proactive and can take the initiative, long running to learn over time, and adaptive to the user's changing interests and needs [Shneiderman and Maes 1997]. She argues that the amount of data is just too much for people to manage with even the best information visualization and direct manipulation systems (which is clearly even more true today than it was in 1997).

This debate highlights a key consideration for intelligent interfaces, discussed in the next section (18.9)—how to keep users aware of what is happening and in control. The general lack of success for AI "expert systems" in the 1980s and the failure of Clippy in 1997 (Section 18.5) didn't help the cause of intelligent interfaces. Today, one worry is the "AI apocalypse," where AI systems become so intelligent they decide to eliminate people, possibly triggered by the "technological singularity" (also called the "AI singularity" or just "the singularity"), which is the point at which machines become so good at learning on their own and improving themselves that they will no longer need human intervention, so the technological growth becomes uncontrollable and irreversible [Vinge 1993]. Luminaries who have warned about this include Vernor Vinge, Stephen Hawking, Bill Gates, Elon Musk, and Ray Kurzweil.

There are many current concerns with the ubiquitous use of *large language models* in AI systems and conversational agents due to their susceptibility to biased and inaccurate outputs (including "hallucinating" data that isn't there), lack of user

control, unclear legality with respect to copyrights on the data used to train the models, and many other factors.

A more immediate concern arises from the algorithmic and machine learning user interfaces discussed in Section 18.6. There are many different dimensions to these concerns:

- *Required level of accuracy*: The required level of accuracy of the algorithm's prediction obviously varies depending on the seriousness of the outcome—if the user doesn't actually like a proposed movie, they may waste a few dollars and a little time, whereas an algorithm predicting to whom to give a home loan can have much more serious consequences. But even in the first case, if the system is rarely right, then users will lose trust in the predictions and ignore them. To the extent that a business model depends on users engaging with the predictions, as in the case of predicting ads, then accuracy correlates with the company's income.
- *Algorithmic bias* is when the results of the algorithms have unfair or incorrect outcomes that favor one group over another. Since many of these algorithms are taught based on a large number of examples, when the examples are skewed, then the algorithm's selections will be as well. For example, algorithms for face and emotion recognition are trained on images from the open web, which are mostly grownups and are mostly shown being happy, so a task of identifying children's emotions, especially negative emotions (sadness, fear), is much less accurate [Bryant and Howard 2019]. More consequential would be algorithms biased by a training set that has a disproportionate number of entries of one group, such as the number of Black men in jail, used in facial recognition from surveillance cameras [Introna and Wood 2004].
- *Privacy*: These algorithms tend to work better when they have more data, leading to a desire by companies to collect more data about people, and to correlate that data with other data, to create ever more detailed profiles of people. This can lead to significant loss of privacy. For example, in order to provide more targeted ads, a company may want to know everything a user has bought no matter what website they purchased it from, requiring tracking of all the user's purchases. A vendor might want to combine the online shopping data with real-world shopping data, such as based on credit card usage and food market cards. A political candidate who wants to advertise might want to know the user's political leanings, which might be apparent from what websites they visited (not just where they purchased things) or even from county voting registration lists. Although Meta CEO Mark Zuckerberg candidly announced that he believed the age of privacy is

"over" [Bosker 2010], there are increasing calls for governments to regulate what companies can collect, how they can use it, who they can transfer it to, and how users can check, correct, and delete information that companies hold. For example, the General Data Protection Regulation (GDPR) was passed by the European Union in 2016 for this purpose.

- *Surveillance capitalism*: A consequence of Internet companies (like Google, Meta, Twitter, etc.) making money from advertising, and of wanting more and more data about people to target those ads more accurately, is that the economic model is quite different than conventional capitalism. When users buy physical products, the customer is the user of that product. However, with Internet companies, the customer is the advertisers, and the product sold to advertisers is the user. This has been called *surveillance capitalism* that, besides the loss of privacy, also "challenges democratic norms and departs in key ways from the centuries long evolution of market capitalism" [Zuboff 2015].
- *False information*: Most of the algorithms and systems were designed under the assumption that people would behave honestly, but events over the last few years have shown the problems with that approach—false and provocative information ("fake news") is often propagated more than true information, and the *micro-targeting* enabled by the data collection can make it difficult to monitor. Given the pervasiveness of platforms like Facebook, Twitter, and more targeted forums, such false information has been said to be "a threat to whatever remains of democracy in the US" [Tucker 2020].
- *Over-trust*: Another issue that affects *all* intelligent interfaces is the well-documented tendency of people to trust what the system is proposing even if it isn't what the user originally desired. We saw this in my early Peridot system (Section 18.7) where users would routinely answer "yes" even when the system was wrong. It is generally much easier for users to accede to what the system proposes than to figure out how to correct it. Sometimes this is called "overtrusting" of technology [Wagner et al. 2018], and it can undermine the user's intentions and make it more difficult for designers to provide users with the appropriate feedback and control.

Research and practice that attempts to address all of these issues is sometimes called *responsible AI*. One aspect is helping users understand what the AI is doing and why it produced the results, which is sometimes called *explainable AI*, but just providing explanations does not necessarily go far enough if the user isn't able to control what the system is doing.

Ben Shneiderman's new (2022) book goes beyond "responsible" or "explainable" AI to argue for *human-centered AI*. This book "offers an optimistic realist's guide to how artificial intelligence can be used to augment and enhance humans' lives." The book argues that being human-centered will reduce "AI's existential threats and increase benefits for users and society..." by creating products that are "reliable, safe and trustworthy" and preserving human control [Shneiderman 2022].

18.9 Summary of Considerations for Intelligent Interfaces

Considerations for using intelligent user interfaces have been highlighted throughout this chapter, and the important ones are summarized below.

- *Usefulness*: Make sure the intelligence is doing something useful for people, beyond what could be achieved using conventional direct manipulation interfaces.
- *Appropriate feedback and control*: Make sure users can understand what the system is doing and what the results are, and how to affect the results if they are incorrect or otherwise not what the user wants.
    - *Appropriate confirmation*: A subtopic is how the system allows the user to provide confirmation, which may be:

        Wait for the user to ask for help, so only the user can take the initiative (e.g., spell checking on demand).

        Provide lightweight indications of what the system wants to do and let the user click on them to request actions (e.g., wavy underlines, auto-fill/autocomplete proposals).

        Always ask the user before taking any action, either using textual questions or popup dialogs.

        Just do the action, and provide clear visibility about what happened so the user can undo it if incorrect.
- *Accuracy*: Research and experience has shown that providing appropriate feedback and control is not enough—people will still "over-trust" the system, so it needs to be as accurate as possible. This also applies to avoiding explicit "fake news" entered by bad actors on purpose.
- *Discoverability and affordances*: Make it clear to users what the system can do and what kinds of inputs users can provide.

- *Appropriate design* so it isn't annoying. This includes good usefulness, feedback, and control, but also good design of the interface, using classic HCI principles and methods (e.g., avoid spurious motion like Clippy).
- *Learning and customization*: It is better if the system learns what the user has already done and if the system is customized to what this particular user will find helpful. Learning from the community over time so the system continually improves in general is also beneficial.
- *Privacy*: Users should be aware of what information about them is being collected, and whether the data collected about them is protected appropriately. Users should have the ability to review and correct any erroneous data about them. Economic models should not be based on activities that harm individuals.

18.10 Chapter Conclusions

Although intelligent interfaces have been studied since the beginning of computers, until recently there were very few successful ones deployed. In the last few years, however, AI has permeated many forms of human–computer interaction and will become ubiquitous. Interaction techniques need to catch up and become more pervasive and standardized so that users will have appropriate and familiar ways to use, understand, and control the intelligent interfaces of the future. Whereas speech seems like the ultimate UI, I believe that we will always need GUIs and multimodal interactions since some things, like long lists, visualizations, graphics, and so on, are much quicker and easier to comprehend visually, and some actions, like adjusting a volume, are much easier to do by direct manipulation (rather than saying "louder, louder, louder, no softer, softer, ...").

PART III

USING, DESIGNING, AND IMPLEMENTING INTERACTION TECHNIQUES

Part II related the stories of many different kinds of interaction techniques. In this part, we focus on more "meta" issues, about interaction techniques in general. Much of this part is focused on programmers who will need to write code to use interaction techniques or to make new ones work. Interaction designers and graphic designers, even if they are not programmers, might need to know some of the material in this part to help evaluate existing interaction techniques or design and evaluate new ones.

Virtually all user interfaces are made out of interaction techniques, so everyone designing or implementing a user interface of any kind will be using the interaction techniques discussed in the previous chapters. This part explains how that is done. Also, since most UI design is creating or using interaction techniques, the topics discussed in this part are mostly relevant to UIs in general and not exclusively for interaction techniques.

As mentioned in Section 1.1, new interaction techniques are being invented all the time, and Section 1.7 (and throughout this book) explains why new ones are needed. Section 1.6 explains why new techniques can have a high economic value, so the topics in this part will hopefully be of value to people engaged in these pursuits.

CHAPTER 19

Formal and Informal Models, Metaphors, and Patterns for the Design of Interaction Techniques

19.1 Introduction

This chapter focuses on ways to describe aspects of interaction techniques at a more abstract level. The topics in this chapter are a somewhat eclectic mix of different ways of thinking about the design of and design with interaction techniques. Some of these techniques might be considered "formal," which means they have a mathematically rigorous underpinning, whereas others are informal. In all cases, the goal is to help people think more systematically about interaction techniques and, in some cases, to offer helpful ways to teach end-users about their use as well.

Although the techniques in this chapter are focused on *designing* interaction techniques or designing UIs that use them, the next chapter (Chapter 20) focuses on how programmers can *implement* user interfaces that use interaction techniques or how the interaction techniques themselves are implemented internally. This distinction is not easy to make in some cases since the models in this chapter might be used to *describe* the implementation as well as the user interface, and such models are mentioned in both chapters.

Other motivations for considering interaction techniques at a higher level include helping to understand the space of possible interaction techniques, which may identify opportunities for new ones if there are unexplored areas of the space. Often these models can provide a precise description of how interaction techniques work, which can be analyzed for errors or performance.

The structure of the previous chapters of this book provides one organization of the space of all interaction techniques, which others may disagree with. For example, I chose to include all kinds of buttons in Chapter 7 about "menus," and

I grouped text entry widgets with text entry methods in Chapter 8. This chapter provides a few other ways of thinking about possible ways to group the different interaction techniques.

19.2 "Model"

The word *model* is used in a surprisingly large number of different ways in HCI, and consequently in this book (even in this one chapter)! These generally share the sense of a model being an *abstraction that represents the real thing*, but not all uses match that definition. Here are some different ways that the term is used in HCI:

- The *user model* in HCI (as popularized by Don Norman's famous book [Norman 1988]) is *the way a user thinks about a system*, which hopefully matches the way the designer and implementer thought about it. The user model, which is entirely inside the user's head, includes how the user thinks the system operates.
- Some models describe *the way a feature is presented to users*. For example, the *color model* (Section 10.7) defines how the user or programmer can specify a color (e.g., RGB, CMYK, HSV, or HSL), and the *undo model* (Section 15.8) describes how users can undo the previous commands.
- *Software models* are *the way that a programmer should think about the implementation* and often refer to particular parts of the software architecture. For example, the *imaging model* (Section 20.12.5) is how graphics are drawn, the *object model* (usually either a *class model* or *prototype-instance model* (Section 2.4.1)) is how objects work, and the *Open Data Model* (Section 11.11.3) is one way that applications connect to data. Section 20.12 describes a number of other models, such as the *retained object model*, *document model*, *interactors model*, *event model*, and *model-view-controller (MVC) model*.
- Somewhat ironically, the MVC model (Section 20.12.6) just mentioned uses the word "model" twice. The first one (in the name) means *the data structure that the user interface is connected to*, which might be as simple as a number for a slider or as complex as the complete data structure representing all the characters in document editor.
- Some software tools that create a user interface automatically are said to use *model-based design* [Sukaviriya et al. 1994], where the model is *a high-level specification from which the user interface is generated or from which the implementation is created* (see Section 20.10).

- As mentioned in the introduction, this chapter describes some *informal models* that *help explain the space of options*, such as the *input device models* in Section 19.5.
- This chapter also describes many *formal models* that *provide a mathematical underpinning for analyzing parts of a UI*, including *state transition diagrams* (Section 19.4) and *human performance models* (Section 19.7). Similarly, the *Fitts' Law model* that allows estimating an input device's speed was covered in Section 4.14.1.
- *Language models* are *representations of how words are used in a natural language* like English and are used to predict words in T9 (Section 8.3.3.2) and other text entry methods like Swype (Section 8.3.6.2) and also to design on-screen keyboards (Section 8.3.6.1).
- Similarly, *AI models* are *the way that pattern matching algorithms used by machine learning and other AI techniques work*.
- *3D models* are *computer representations of three-dimensional objects*, which were discussed throughout Chapter 12.
- Sometimes I refer to a *product model*, which refers to *the particular version of that product* (usually a physical device), like the Teletype Model 33 in Figure 2.3.
- Finally, I don't think we have any places in this book where we refer to *fashion models*, who are *people who show off clothing and accessories*.

19.3 Metaphors and UI Design Patterns for UIs

A *metaphor*, as applied to a user interface, is a mapping between the user interface and another situation that is more familiar to a user in order to make the user interface easier to understand [Halasz and Moran 1982]. A *design pattern* is a more general term that refers to a re-usable form of a solution to a design problem, which sometimes, but not always, uses metaphors to the real-world as its explanation. For instance, the *shopping cart design pattern* [van Duyne et al. 2002], used in most eCommerce sites, is a metaphor for physical shopping carts in a supermarket. Note that this use of *design pattern* is focused on what the user sees and is not the same as a *software pattern* for code discussed in Section 20.12 since those are used by programmers and hidden from users.

Note that in English, officially, an analogy is saying something is *like* something else ("this screen is *like* a shopping cart"), whereas a metaphor is saying something *is* something else ("this list *is* your shopping cart"). In this book, following the usage in the technology community, we are not making a distinction

between the two terms and use them interchangeably, with "metaphor" being the more popularly used term.

The classic example of a metaphor is the *desktop metaphor*—the computer is a desktop, already extensively covered in Chapter 11. That metaphor, which was invented for the Xerox Star, released in 1981, has a number of sub-pieces that help reinforce the metaphor and explain lower-level parts. For example, the computer file system is like the files and folders in a physical filing cabinet, and deleting files is like throwing them into a trashcan (the trashcan metaphor was actually not part of the Star but was added with the Apple Lisa, Section 2.4.7). At least one system took the metaphor even further, presenting a pretend 3D environment (Figure 3.13) where, for example, the user's contact list is presented as a Rolodex. Chapter 11 also covered some alternative metaphors for the user's whole environment, like the rubber sheet metaphor (Section 11.11.1) or the timeline metaphor (Section 11.11.2, which isn't a metaphor to something physical but rather to a different abstract construct).

Other metaphors that have already been covered in this book include the *tool metaphor*, (Section 10.4) where operations in the palette are first picked up and then used, like a carpenter's tools from a toolbox, and the *boxes and glue* metaphor (Section 9.9), where items to be laid out on a page are thought of as boxes attached to each other by stretchy glue. The *Rooms* metaphor for organizing collections of windows was described in Section 11.10. Having tabs like a notebook, or having the whole computer pretend to be a notebook, was a feature of HyperCard (Section 2.4.9), PenPoint (Section 3.5), and regular tabbed dialog boxes (Section 9.4). Cut-and-paste was invented as a metaphor for how physical pieces of paper were cut and pasted in page layout (Section 23.5). Actually, almost everything in a computer, including most of the interaction techniques, has a name that is an analogy to something that existed before: mouse, tablet, notebook (computer), menu, button, radio button, scroll, thumb, knob, window, icon, document, calculator (application), agent, and so on, which might be considered to be metaphoric.

Most *design patterns* are not metaphors. The book *The Design of Sites: Patterns, Principles, and Processes for Crafting a Customer-Centered Web Experience* [van Duyne et al. 2002] lists 107 patterns for the web organized into 13 groups, only a small number of which are metaphors (like the shopping cart pattern).

While metaphors can help people understand something unfamiliar, they can also be limiting. For example, researchers long ago argued that novice users may be limited in not learning aspects of the system that are different from the metaphorical system, or they may make mistakes by carrying over incorrect aspects from the other system [Halasz and Moran 1982]. For example, physical files and folders

have no concept of "shortcuts," which are links to the original file that can be put in other places to provide multiple paths to the same file, so a user limited to only the metaphor would never learn about shortcuts. On the other hand, physical folders can get full, so bringing that aspect over to computer folders would also be limiting to users [Halasz and Moran 1982]. However, as people are much more familiar with computer UIs compared with 1982, this worry has dissipated.

19.4 State Transition Diagrams

At the other end of the formality scale are *state transition diagrams* (STDs), also called state transition models, state transition networks, finite state automata, or just state diagrams or transition diagrams, which were briefly introduced in Section 1.3.7. (These terms actually have slightly different meanings and features in mathematics or formal Computer Science theory, but they are close enough for the kind of analysis discussed here.) A state diagram is a finite collection of states, usually represented as circles or rectangles, also called *nodes*, connected by a finite number of directed edges, called *transitions*, between the circles, usually represented by arrows, thus forming a directed graph. In user interfaces, the states usually represent modes of the user interface or interaction technique, and the transitions are usually user inputs, like using a mouse button or clicking on a link, which results in the user interface changing modes. For example, in Figure 19.1, a mouse-down event over an object causes that object to switch to dragging mode, where it follows the mouse, which stops when the mouse button goes up. State transition diagrams have been used by many systems to describe UIs [Newman 1968b, Jacob 1985, Buxton 1990, Olsen Jr. 1992], including a research system

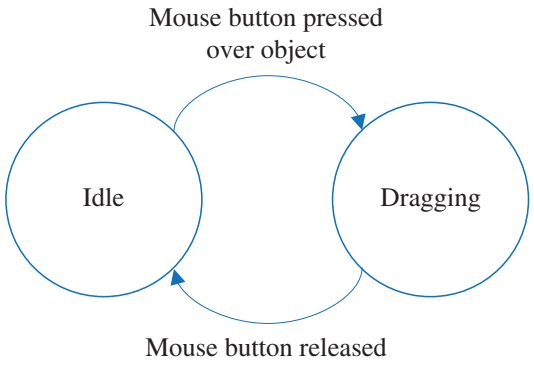


Figure 19.1 A simple state transition diagram with two states and two transitions. Source: Original by author.

called InterState from 2014 that allows users to draw STDs to defined interaction technique behaviors (see Figure 20.8).

A key advantage of a state diagram is that it clearly shows what can happen in a user interface and what is supported. The assumption is that inputs that aren't shown do nothing, so the system of Figure 19.1 would ignore all other inputs. Obviously, this is pretty useless, so real state diagrams would be more comprehensive, as in Figure 19.2, which is a somewhat more complete description of the behavior of a checkbox, like in Figure 1.3, that supports hover effects and also pressing, moving outside, and releasing as a way to abort the operation. This is still not a complete description of the behavior of Figure 1.3 since we would need to add states and transitions to support all the other behaviors described in Sections 1.1 and 7.3.11.1. Note that the abort behavior supported by hitting the Esc keyboard key (explained in Section 7.3.5) highlights a limitation of conventional state diagrams—an arrow labeled with Esc would need to be added to *every* state since the user should always be able to abort, no matter the current state, resulting in a mess of extra wires. Some have argued for augmenting the diagram with a "pervasive" transition that can always be used to address this problem [Olsen Jr. 1992, p. 53].

However, there are more serious problems that limit the applicability of this approach—state diagrams don't support activities that can be performed in any order (e.g., items in a form can be filled in any order) or even interleaved (e.g., the user can partially fill in one field, go to another field and fill that in, and then return and finish the first one). Many extensions of transition diagrams

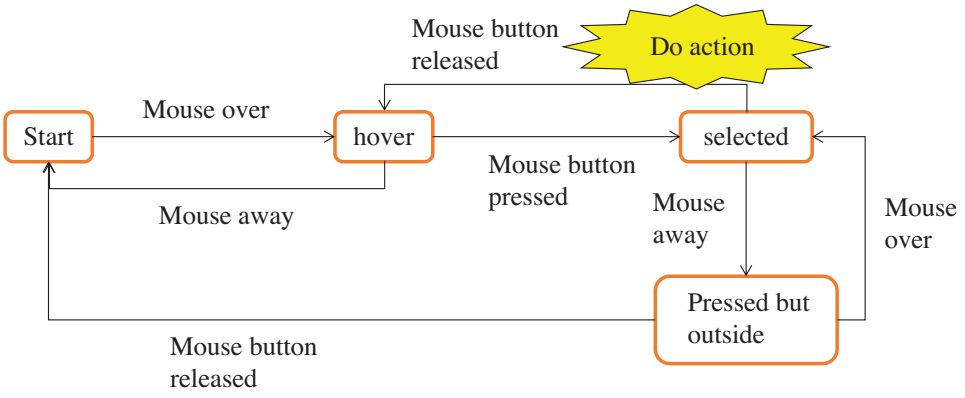


Figure 19.2 Somewhat more complete state diagram for a control such as a checkbox. Source: Original by author.

have been developed to address some of these problems, which make the models more complex to learn and to use, but more expressive. For example, David Harel invented *StateCharts* that provide for parallel and optional activities, inheritance, and so on [Harel 1988], which have been adopted as one of the Unified Modeling Language diagrams.

Still, regular state diagrams like Figure 19.2 have proven very useful for helping designers and users reason about and convey all the options that need to be handled by an interaction technique, and avoid forgetting about things that can happen (e.g., forgetting to deal with the button released event while the mouse is outside in Figure 19.2, mentioned in Section 1.8.1). As explained in Section 20.12.4, my group found state diagrams very useful in designing and explaining the *interactors* model.

Section 20.8 discusses using state diagrams to actually *implement* a user interface or an interaction technique beyond just using it as a specification tool.

19.5 Input Device Models

One kind of model that is particularly useful for thinking about interaction techniques involves trying to map out the space of input devices and what kinds of values they can be used to input. For example, previous chapters have discussed many ways to do pointing (Chapter 4), scrolling (Chapter 5), selection (Chapter 7), and text entry (Chapter 8), but is there a more systematic or formal way to discuss the space of all of these options? Next are some attempts.

19.5.1 Foley and Wallace (1974) and Chan (1984)

One of the earliest attempts was by Jim Foley and Victor Wallace in 1974 [Foley and Wallace 1974]. At the time, there were a wide range of devices that might have been available to users: "The diverse devices available at a given terminal may include typewriter keyboard, function keys, lightpen, tablet, joystick, and dials" [Foley and Wallace 1974, p. 466], and each user might have a quite different set of these (see also Section 2.4, Chapter 4, etc.). Foley and Wallace wanted to allow software to run with a variety of kinds of devices, so they proposed the idea of *virtual devices* that would be a high-level representation of all of the kinds of input available. They proposed that "each device can be treated as a physical realization of one or several of just four distinct virtual devices" [Foley and Wallace 1974, p. 466]:

- *Pick*: to select among displayed graphical objects.
- *Button*: to represent any physical button, including physical switches (Figure 2.8) and keyboard keys.

- *Locator*: to represent a location on the screen where there is not necessarily something now. This kind of input was available at the time from a tablet, joystick, or mouse.
- *Valuator*: to provide a single, real-valued number, as would come from a knob (Figure 2.8) or physical slider (Figure 5.5, see also Chapter 5).

This *virtual device* concept was very influential and was used by a variety of graphics standards of the time, such as CORE and PHIGS [ANS 1984]. In these standards, which assumed a single-threaded implementation in programming languages like FORTRAN, the programmer would specify which kind of input was desired, and the program would block all other activity while waiting for it (e.g., now select an object, then select an operation, then select a value). However, as direct manipulation user interfaces with their less modal and more user-focused control took over in the 1980s (along with the homogenization of input devices to just a mouse, with the rest of the controls moving to on-screen widgets), this model, especially the separation of pick and locator, was no longer useful.

Ten years later, Foley and Wallace, joined by then graduate student Peggy Chan, defined a model focused on the user's view of "interaction tasks" rather than the hardware level of devices, and they identified the following six types [Foley et al. 1984]:

- *Select*—one or more from a set of objects.
- *Position*—indicate a position on the screen.
- *Orient*—rotation in 2D or 3D space.
- *Path*—series of positions or orientations.
- *Quantify*—a single value, such as a number.
- *Text*—input a text string, for example, by typing it.

These are still higher-level than interaction techniques since "select," for example, is used for menus, selections in a graphics editor, type-in of the name of an item, and so on [Foley et al. 1984].

19.5.2 Buxton's Classification (1983)

Bill Buxton argued against considering input devices at the "virtual device" level in a 1983 paper since from the user's point of view input devices are not interchangeable, and different interactions would be vastly different or even impossible depending on the particular characteristics [Buxton 1983]. For example, accurately tracing a picture from paper onto a computer is easy with a tablet in absolute mode but much more difficult with a mouse. He proposed the taxonomy of Figure 19.3

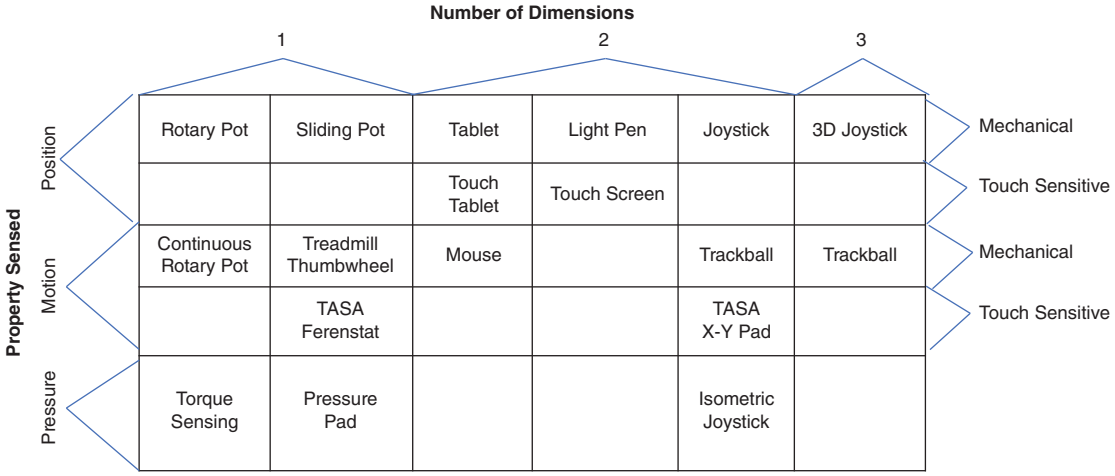


Figure 19.3 Buxton's 1983 taxonomy of characteristics of input devices. Source: Buxton [1983].

to capture the number of dimensions of input (1D, 2D, or 3D input), the property sensed (position, motion, or pressure), and whether it was mechanical or touch sensitive. In each cell are the physical input devices that work that way.

19.5.3 Card, Mackinlay, Robertson Model (1990)

Stu Card, Jock Mackinlay, and George Robertson together are famous for a collection of ways to use 3D representations to visualize data, such as *cone trees* and the *perspective wall* [Robertson et al. 1993]. As part of the early work on that, they investigated the space of input devices and how those might be used for 3D manipulations [Card et al. 1990]. This resulted in the design space of Figure 19.4, which generalizes the models of Foley, Wallace, & Chan (Section 19.5.1) and Buxton (Section 19.5.2) to focus on the device's expressiveness. It shows (see Figure 19.4) the way the user moves the device (linear or rotary), the dimensions (X, Y, Z) based on how many icons (e.g., the mouse is shown with two hexagons for X and Y), whether it is continuous or discrete (which vertical column it is in, based on the log scale at the bottom—1, 10, 100, or infinite values), and whether it is absolute (P) or relative (dP) or measures force (F) (see also Section 4.12).

19.6 Pragmatic, Lexical, Syntactic, Semantic, and Conceptual Level Model

As discussed in Section 2.2.1, the original computer users were all programmers, so it is not surprising that early thoughts about how to think about user interfaces

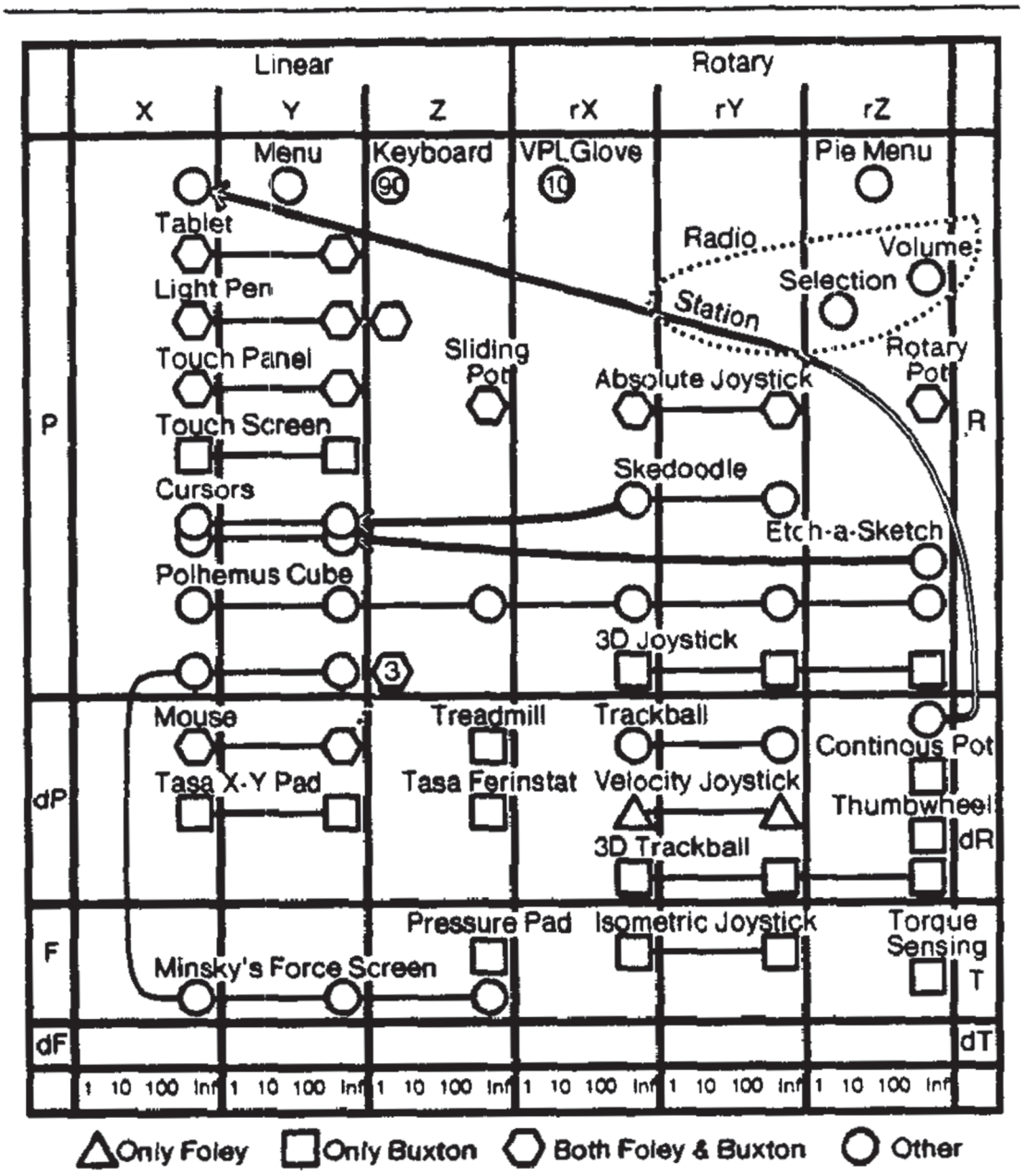


Figure 19.4 Card, Mackinlay, Robertson Design Space (1990). The shapes reference the Foley & Wallace (Section 19.5.1) and Buxton (Section 19.5.2) classifications. Source: Fig. 4 of Card et al. [1990].

were derived from compiler theory (a *compiler* is a software program that converts a program's code into a binary form that can be executed by a computer). Compilers generally have multiple passes. First, the *lexical* pass breaks up the sequence of characters of the code into *tokens*, which are the strings that have meanings. Typically, there are special characters like +, *, (), [], words like "if," "void," "begin," "getElementByID," "x," and spaces and tabs as separators. The *syntax* level tries to *parse* the string of tokens to identify if they are in a correct order (see also Section 10.2). The parser also associates operations with their parameters (e.g., identifying what should be added when a "+" token is in the code). Next, the *semantic* pass identifies the detailed type of each token and how to create an efficient binary format from the results of the parse.

In their classic 1982 graphics textbook,[1] Jim Foley and Andy van Dam use these steps as the basis for their "language model" of the user–computer interface [Foley and van Dam 1982, p. 220], adding one more step at the high end, called "conceptual." The *conceptual level* describes the key concepts in the interface from the user's point of view, including the key kinds of data and their operations and relationships. This forms the user's mental model of the UI. For example, in a text editor, the conceptual level would be the characters and the files they are part of, with operations like insert and delete. Going down the levels, the *semantic level* in this model is the functionality of the system—what it can do. This idea of separating the semantics (*what* can be done) from *how* the user makes that happen has been an enduring goal of most software models and tools. The *syntactic* level is the sequence of inputs and outputs for operations. For the input, it is the grammar defining the legal sequences of tokens. For output at the syntax, Foley and van Dam include the "spatial and temporal factors ... [of the] 2D and 3D organization of a display" [Foley and van Dam 1982, p. 220], but that proved not to be clear or helpful. Finally, the *lexical* level defines how the tokens are actually formed and differentiates, for example, invoking a delete operation using the Delete key on a keyboard, from selecting a "Delete" item in a menu, from clicking on an "X" icon in a ribbon or palette.

Bill Buxton argued in 1983 that one needs an even lower level, which he calls the *pragmatic* level, to capture the "issues of gesture, space and devices" [Buxton 1983], including how the physical input devices work, the required actions to make

1. This was discussed in a *graphics* textbook since from the computer science side HCI grew out of the graphics community and much of the early work was published in the SIGGRAPH conference (which started in 1974) before the SIGCHI conference started (in 1984). My first conference publication was at SIGGRAPH in 1983 [Myers 1983]. Other early influences on HCI came from psychology and human factors (see also Section 1.3.6).

the input (e.g., press down and hold a mouse button vs. click-move-click), and the ergonomics.

Although the overall model is quite old and doesn't make much sense when applied to the *output*, it can still be useful when thinking about low-level issues of a UI, especially with respect to the pragmatic–lexical–syntax levels. For example, Section 10.2 discusses the *syntax* for operations, such as prefix (operator first—like "delete *file1*"), postfix (operator last—like *selected-text* turn-bold), and infix (operator in the middle—like "3+4" or *selected-circle* change-color *blue*). The pragmatic and lexical levels have been extensively discussed with each interaction technique in the previous sections.

19.7 Human Performance Models

A totally different kind of "model" are mathematical models that can be used to predict how long a user will take to perform tasks without needing to run any user tests. These use a "model" of how people work, which has been validated to be reasonably accurate, given certain assumptions. They tend to work best for low-level, expert tasks that do not require the user to do much, if any, problem-solving. For example, the Fitts' Law model, described in Section 4.14.1, provides a mathematical way to predict the speed of using a pointing device, based on properties of the target to click on (how big and how far away) and of the device, assuming the user doesn't have to think about where to click. This section discusses two more models: the keystroke-level model (KLM) and goals, operators, methods, and selection rules (GOMS).

19.7.1 Keystroke-level Model

Cognitive psychologists and other scientists have long studied how long it takes for humans to perform various tasks. This includes how long it takes, on average, to move one's hands (e.g., to point at a target, see Section 4.14.1), and also how long it takes to perceive and understand something that one is looking at, how long it takes to store or retrieve a piece of information to short-term or long-term memory, and how long to think about and make low-level decisions. In a classic early work, Stu Card, Tom Moran, and Allen Newell developed the *Model Human Processor* (Figure 19.5) to summarize all of these values. They also created a mathematical model, called the *Keystroke-Level Model* (KLM) that calculates how long it will take an expert to perform tasks on a computer, taking into account all the low-level actions required [Card et al. 1980a]. KLM includes four different physical motor operations: **K** for keystroking time (based on average typing speed, see Table 8.1), **P** for the time to point to a target, based on an average time as predicted by Fitts' Law (Section 4.14.1), **H** for the "homing" time to move one's hands between the

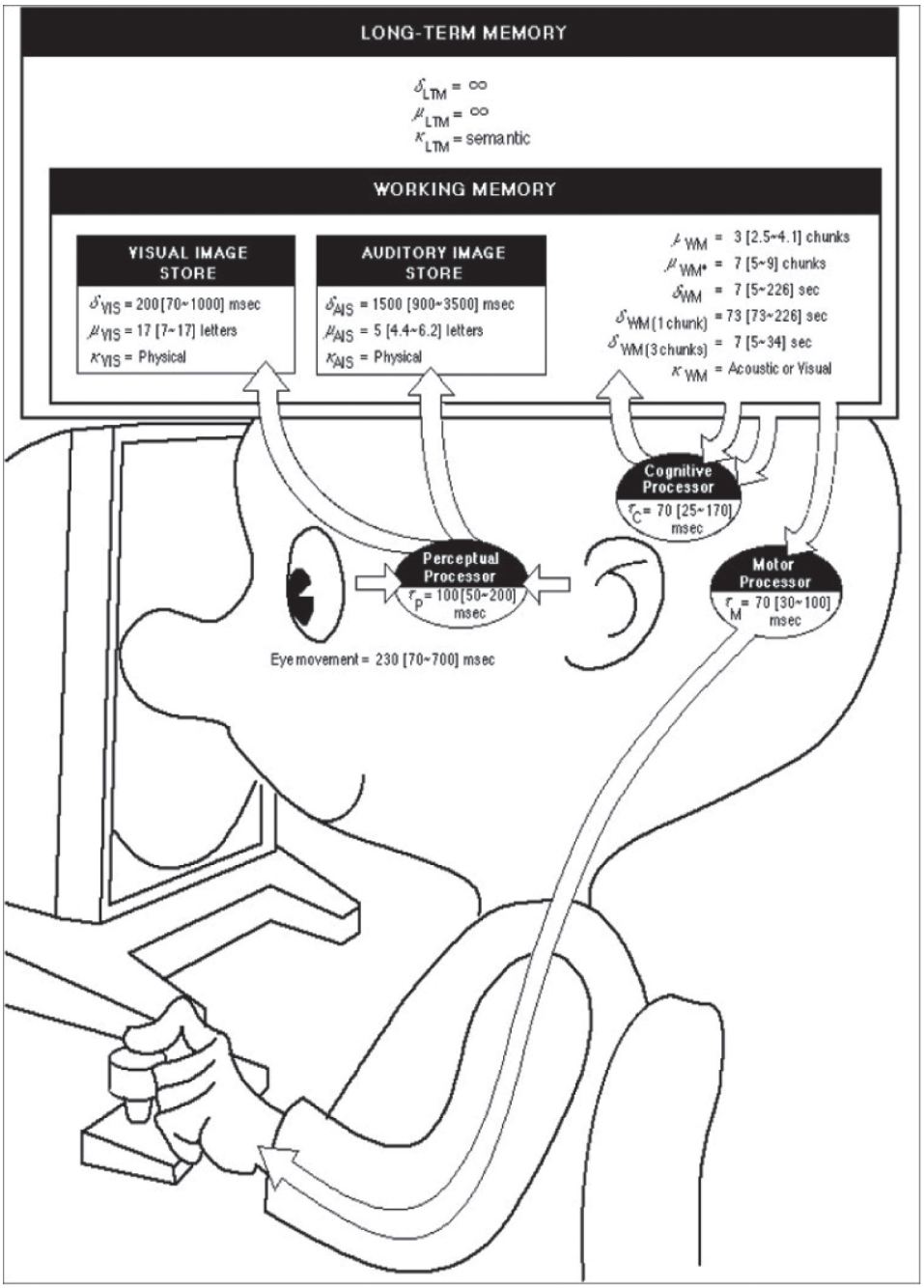


Figure 19.5 Memories and processors of the Model Human Processor and their time constants. Source: Fig. 2.1 of Card et al. [1983]. Reproduced with permission.

keyboard and the mouse, and **D** for drawing lines with the mouse, which is basically never used. KLM also includes **M** for the mental thinking time to perform actions and **R** for the system's response time (when the user has to wait for the system). KLM includes estimates for the lengths of these operations, and heuristics for when to insert the **M** (thinking) operator. Then, one only has to add up the number of operations of the specific types that are required to perform a task to get an estimate of how long it will take, as shown in Figure 19.6. One can create such calculations for different sequences that might be used in different applications or with different interaction techniques to compare the times [Card et al. 1980a]. Studies showed that the times from KLM were pretty close to the average actual measured times for expert users.

19.7.2 Goals, Operators, Methods, and Selection Rules

One might notice that there are many time constants in Figure 19.5 that are not used in the simple KLM model, such as perception, memory, and deeper thinking times. Card, Moran, and Newell went on from KLM to define a more comprehensive model called *GOMS*, which stands for Goals, Operators, Methods and Selection rules, in order to be able to create models of more complex human activities at the computer [Card et al. 1983]. *Goals* are what the user wants to achieve and are expressed at various levels of detail, so that EDIT-MANUSCRIPT would have subgoals for each of the individual editing tasks needed, such as MOVE-TEXT. The *operators* are the low-level operations that the user does, similar to the ones for the KLM, such as pressing keyboard keys and moving the mouse. The *methods* are the intermediate-level ways to achieve a goal. For example, for MOVE-TEXT, the user

Description	Operation	Time (sec)
Reach for mouse	**H**[mouse]	0.4
Point to word	**P**[word]	1.1
Select word	**K**[YELLOW]	0.2
Home on keyboard	**H**[keyboard]	0.4
Call Replace command	**MK**[R]	1.35+0.2
Type new 5-digit word	**5K**[word]	5*0.2
Terminate type -in	**MK**[Esc]	1.35+0.2
Total time		**6.2**

Figure 19.6 Times to perform an editing task as predicted by KLM for a particular editor. **H** is the time to home to the mouse or to the keyboard. **P** is the time to point. **K** is the time to hit a keyboard key or a mouse button (YELLOW was the middle mouse button), where 5**K** means 5 keys. **M** is the mental operation (here followed by a K operator). Source: Original by author, based on data in Card et al. [1980a].

might use a CUT-PASTE-TEXT-METHOD or else a DRAG-DROP-TEXT-METHOD. When there are multiple methods, the *selection rules* provide IF–THEN rules for picking which one to use, such as: "IF the source and destination positions for the move are both visible on the screen, THEN use DRAG-DROP-TEXT-METHOD, ELSE use CUT-PASTE-TEXT-METHOD." Once a GOMS model was fully defined, then, like KLM, an automated process could calculate the estimated time for a user to perform tasks.

For both the KLM and GOMS models, there were eventually many variants and extensions [John and Kieras 1996] that would handle more nuanced analyses of more complex systems. For example, GOMS was extended to handle tasks that involve visual searching—to have operators for how long it might take for users to find items of interest [John and Kieras 1996]. These extensions were typically embedded in programming environments such as AI "expert systems" or "production systems" like *ACT-R* [Anderson et al. 1997] or *SOAR* [Baumeister et al. 2000], which were already designed around IF–THEN rules like the GOMS selection rules.

A typical analysis of an application, like a text editor, would involve hundreds of the entries for either KLM or GOMS, so it proved daunting to author. Therefore, Bonnie John and colleagues developed *CogTool* (Figure 19.7) where the designer could enter screenshots of all the screens and show storyboards about how tasks were executed by marking the transitions among the screens, along with the other required operations, and it would create the model and output the estimated time

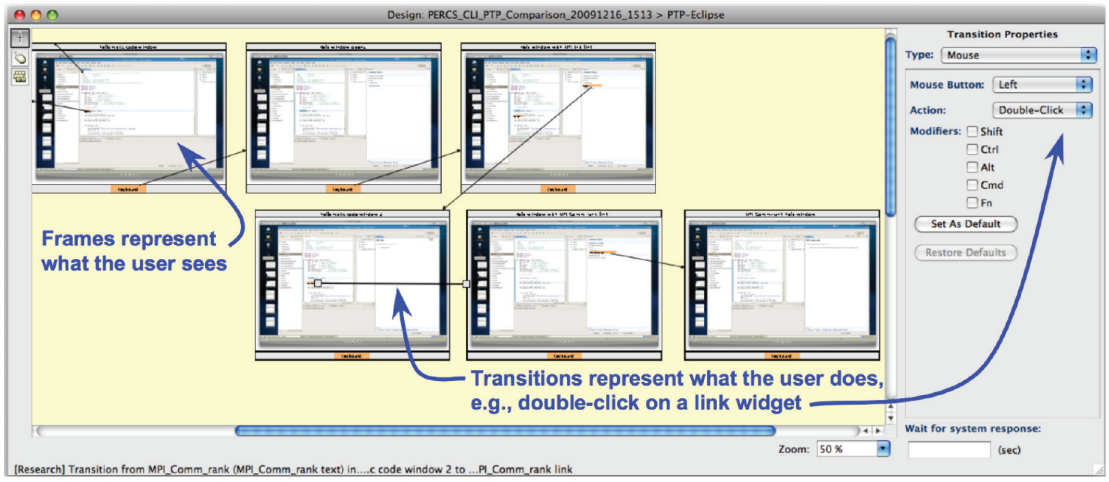


Figure 19.7 CogTool. At left are some of the screens ("Frames") of the Eclipse application, with arrows ("Transitions") showing what the user does to perform a particular task. Source: Fig. 1 of Bellamy et al. [2011].

for that task performed in that way [Bellamy et al. 2011]. Since the application being analyzed did not necessarily need to be implemented yet—the screenshots could be of prototypes—this could help designers estimate which of several alternative designs would be the most efficient.

19.8 Other Models

There are a wide variety of other formal and informal models that have been used with interaction techniques. In some sense, any representation that comes out of any kind of user interface requirements or design method [Hartson and Pyla 2012] can be called an informal "model" of the design—the models that result from a *contextual inquiry and analysis* (flow, cultural, artifact, sequence, and physical models) [Beyer and Holtzblatt 1998], sketches, storyboards, affinity diagrams, information architecture diagrams, lists of requirements, journey maps [Howard 2014], use cases, and so on, are all representations of user interfaces at some level.

One that was alluded to in the previous section is *hierarchical task analysis*, which deserves a little further explanation. One step in performing a GOMS analysis is to break the task up into a hierarchical sequence of goals or tasks to be performed. This hierarchical tasks analysis is also useful on its own and has been mentioned in virtually all UI textbooks [Foley and van Dam 1982, Shneiderman 1987, Hartson and Pyla 2012]. This tree of tasks can be independent of the particular way the user interface performs the tasks, at least at the higher levels, but at lower levels, as discussed in Section 19.7.2, the tree can include specific actions in a user interface to accomplish the tasks. Whereas knowing the users' tasks is vital to supporting them, we have found that creating a detailed hierarchical task analysis model is difficult and usually not effective as a way to represent the needed information, so we teach students in our HCI courses at Carnegie Mellon University to use other models instead, like journey maps [Howard 2014] or the models that are part of a contextual analysis [Beyer and Holtzblatt 1998].

19.9 Considerations for Models

If one was trying to create a new interaction technique, it might be useful to use some of the models described in this chapter. For example, STDs can be useful to make sure that important possible transitions are not forgotten in the implementation. It is possible that a new metaphor might help designers and end-users better understand the interface, but there are problems with that approach [Halasz and Moran 1982] since the metaphor may be misleading in various ways. The input device models may help a researcher better understand the design space, and the input parsing levels (pragmatic, lexical, syntactic, semantic, and conceptual) may help researchers separate different aspects, but these do not seem generally useful.

If one is concerned about the maximum performance that can be expected for experts with the user interface, then the human performance models may be useful, but since these models do not account for problem-solving or creative exploration, their applicability is limited.

19.10 Chapter Conclusions

This chapter focuses on models, metaphors, and patterns for the UI designer or end-user. Whereas some metaphors, like the desktop, tool, and shopping cart metaphors, have been quite successful, there are not a lot of other examples. There are more design patterns that have been widely used, but again, it seems like a relatively small set.

For formal and informal models, however, it is possibly not too surprising that these have not caught on for designers and end-users of interaction techniques since those models have not caught on for HCI and even computer science in general. The existing attempts have mostly been narrowly focused and thereby missed important aspects, making them less compelling. The next chapter includes discussions of models and patterns for the UI *implementations* that have been somewhat more successful.

CHAPTER 20

Tools and Models for Using and Implementing Interaction Techniques

20.1 Introduction

The previous chapter focused on models and patterns for *designing* interaction techniques, or designing with them. This chapter, in contrast, focuses on *implementing* user interfaces and interaction techniques, and hence will primarily be of interest to programmers (although Section 20.8 is relevant to non-programmers who are UI designers).

It has long been known that user interfaces are particularly difficult to both design *and implement* [Myers 1994], as covered in Section 20.2 below. There have been a wide variety of techniques that are relevant to the implementation, including various layers and organizations of software libraries (*toolkits* and *frameworks*, Section 20.4), object class hierarchies for those libraries (Section 20.5.3), ways to pull out the text and related information into *resource files* (Section 20.7), interactive and graphical tools including prototypers (Section 20.8), automatic creation of UIs from models (Section 20.9), and various models and patterns to organize the software (Section 20.12). As mentioned in Section 2.5.2, virtually all user interfaces today are created using the kinds of tools discussed in this chapter, and since most programming involves accepting from or displaying information for users (except possibly for embedded or low-level algorithm programming), UI tools impact most programmers and designers [Myers et al. 2000a].

There have literally been thousands of commercial and research tools over the years that have tried many different approaches to making the implementation and use of widgets easier, so it is clearly impossible to cover them all in this chapter. Even considering just the "important" ones, there have been whole books about early user interface software [Myers 1992b, Olsen Jr. 1992, Bass and Dewan 1993], and commercially, there were a wide range of GUI Builder tools, often targeted at

Unix or Windows machines, or cross platform. There were also tools targeted at the special-purpose devices, like for creating Palm Pilot UIs. Between 1994 and 1997, I maintained a list of UI tools (which won a few web awards) and identified about 100 different examples (https://www.cs.cmu.edu/~bam/toolnames.html), but virtually all of these companies went out of business due to their target platform disappearing or because the compiler vendors included similar functionality into their Integrated Development Environment (IDE) for free, like Microsoft Visual Studio's builder. For my Advanced User Interface Software class in Spring 2017, I had the students help make an updated list (http://tinyurl.com/ToolList2017),[1] and we again found over 100 tools, most of which were targeting building web UIs.

Therefore, this chapter focuses on some principles and examples most relevant to interaction techniques and highlights a few examples of various kinds.

20.2 Why Hard to Implement

As I said in the very first issue of ACM *Interactions* magazine, user interfaces are especially hard to *implement* because [Myers 1994]:

- They are hard to design, requiring iterative implementation.
- They are reactive and must be programmed from the "inside–out."
- They generally require multi-processing.
- There are real-time requirements for handling input events.
- The software must be especially robust while supporting aborting and undoing of most actions.
- It is difficult to test user interface software.
- Most programming languages do not provide support for user interfaces.
- The tools to help with user interfaces can be extremely complex.
- Programmers report an added difficulty in modularizing user interface software.

These conditions still exist today, and, if anything, there are increased complications such as that the UIs often must be implemented using *multiple* complex languages and tools—for example, using JavaScript, HTML, and CSS for the front end that communicates with a backend that must be developed in JSON, Java, and SQL. This chapter summarizes some of the commercial and research efforts at addressing these challenges.

1. Which links to https://docs.google.com/document/d/1qeBkQaoIBmwK9Z7LmWP4L_4GKgddzNn7YmvSQuPukGo/edit#heading=h.3j7u4724qxf9.

Given that interaction techniques are quite hard to *design* and have many states and behaviors (see, e.g., Section 1.1), it is not surprising that they are also hard to *implement* correctly as well. Good examples of these difficulties were the implementations in Adobe Flash of custom widgets in the early 2010s. Flash became the main way that designers were able to provide dynamic and custom interactions on the Web, but there were no widely accepted libraries, and developers created their own implementation of buttons, checkboxes, and scrollbars. Many of these had bugs, especially where the implementers did not correctly handle edge cases like when the mouse button goes down inside the widget, but goes up outside, so the mouse-up event goes elsewhere, leaving the widget thinking the mouse button is down when it isn't (also mentioned in Sections 1.8.1 and 19.4).

Another problem for programmers is the churn of these tools today. For 30 years, from about the 1980s through the 2010s, a programmer could become an expert in one library and have a pretty good career using it for up to 10 years since they changed slowly and incrementally. Examples include the original MacOS toolkit (1984–1997), the revised Mac toolkit based on NeXT (1997 to today, also called Cocoa), the Microsoft Windows SDK (starting 1992 with layers added slowly like Win32 SDK (1995), OLE/COM (from 1993), .Net (from 2003), etc.), PalmOS (1996–2004), the Java JDK (1996 to today), iOS SDK for iPhone/iPad (2008 to today), and Android (2008 to today). However, today, many programmers, especially doing web and app development, need to learn a *collection* of tools across multiple languages, and these tools and languages change dramatically and as frequently as every six months. For example, once popular web front-end development tools and languages that are no longer much used include Java (for the Web), Flash, Ruby on Rails, PHP, ASP, JSP, CoffeeScript, jQuery, the original Angular JS, and many others. Apps and websites built with them need to be reprogrammed. Whereas programs that I personally wrote in C++ and compiled for Windows in 1997 *still run* today on Windows 10, in contrast any web application using a web API typically stops working after a few months since the libraries change in non-compatible ways.

20.3 Definitions

As shown in Figure 20.1, UI software tools can be thought of as spanning several levels. Starting from the bottom:

- The *operating system* provides a set of basic functions such as file and network management and hardware device drivers. Examples of operating systems include Windows 10 for PCs, MacOS for Macintosh computers, Unix or Linux, iOS for iPhones and iPads, and Android for a variety of devices. When considering web applications, the operating system is replaced by the *web*

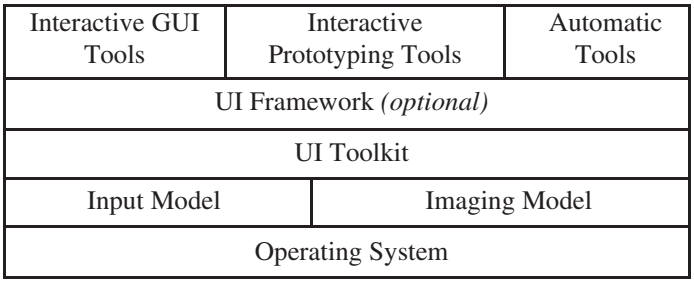


Figure 20.1 Layers of UI software tools. Source: Original. Updated from Myers [1995].

browser, which provides its own set of primitives—for example, through the HTML and JavaScript run-time libraries (which themselves are implemented on top of an operating system).

- The next layer up is the way that low-level input and output are handled. The *input model* controls how user events such as the keyboard and mouse are made available to the code (see Section 20.12.2). Also at this level are how the graphics and text are drawn on the screen, which is usually called the *imaging model* (Section 20.12.5). In previous writings (e.g., Myers [1995]), I have called this layer the *windowing system* since the input and output are usually provided by the window manager part of the operating system, but given the extra layers of code now used between the windows and the UIs, focusing on the input and output models seems more useful.
- The *UI toolkit* is a library of UI widgets that application programs can use. Thus, most of the interaction techniques discussed in this book (scrollbars, sliders, menus, buttons, text entry fields, dialog boxes, etc.) are provided to applications by the UI toolkit layer. Examples of UI toolkits today are the collections of widgets provided by Windows, MacOS, iOS, or Android. For the Web, it would primarily be the widgets provided by HTML.
- Many systems today provide an extra layer of code on top of the collection of widgets, which I am calling the *UI framework*, which makes it easier to write the code for interactive applications, typically by providing a collection of middle-level object-oriented (OO) classes. Most of the patterns discussed in Section 20.12 are supported by various UI frameworks. Popular UI frameworks today are ReactJS, Angular, and Vue for HTML/CSS/JavaScript. Another example is MacOS, which provides a built-in framework that supports the model–view–controller pattern (MVC) (Section 20.12.6) and the document model (Section 20.12.7).

- The collective term I have used to include both toolkits and frameworks is the *application programming interface (API)*, which generally means the collection of software utilized without change to create other software. APIs can be easier or more difficult to use by programmers, giving rise to the study of *API usability* [Myers and Stylos 2016].
- *Interactive GUI tools* support the user laying out and setting properties of widgets interactively, and they create a representation that can be used by the real application. Often these tools will provide a palette of the available widgets, from which the designer can drag-and-drop widgets into the emerging layout. Other names for these tools include *resource editors*, *interface builders*, *GUI Builders*, *or form editors*. Examples include ResEdit, the NeXT Interface Builder, Visual Basic's Form Editor, and many research tools. Often, IDEs like Microsoft Visual Basic, Microsoft's Visual Studio, Eclipse, Apple's Xcode, and Android Studio have a built-in interface builder. WYSIWYG editors for web pages (which can include interactive elements like text fields and menus) such as Adobe Dreamweaver or WordPress also count as interactive GUI tools. There have also been many research systems that have contributed to this area (e.g., Myers and Buxton [1986], Cardelli [1988], Myers [1991], Hashimoto and Myers [1992]).
- *Interactive prototyping tools* also support the user laying out and setting properties of widgets interactively, but the result can only be used for *simulating* the UI and for user testing and cannot be used as part of the real application. There are many examples of historical and current prototyping tools, going back to Dan Bricklin's DEMO (1985) (Chapter 26) and HyperCard (1987) (Section 2.4.9), with modern examples including Axure, Balsamiq, Adobe XD, Figma, and many others.
- *Automatic tools* take a high-level specification of the UI and use algorithms to decide on such features as which specific widgets to use, the layout of where the widgets should go, and properties of the widgets. The high-level specification is sometimes called a *model* of the UI, so these tools are examples of *model-based design* of the UI. This has been an active area of research (e.g., Zanden and Myers [1990], Frank and Foley [1993], Kim and Foley [1993], Szekely et al. [1993], and Sukaviriya et al. [1994]), but there have also been a few commercial examples, such as Apollo's Open Dialog tool from 1987 [Apollo Computer Inc 1987], or tools based on the UIML specification language from www.harmonia.com or based on XIML from www.redwhale.com. HTML might also be said to be an example since it generally describes what is desired and not how to do it or where it goes. Recently, AI

tools including LLMs like ChatGPT has been able to create the code for UIs from a natural language description, which I would also classify as *automatic tools*.

As mentioned in Section 19.2, note that the previous definitions (and the rest of this chapter) use the word "model" in many different ways.

20.4 Why Tools Are Important

User experience (UX) tools are used throughout the design process. Although all types of tools are used during different phases, designers generally use tools that produce more visual representations early in the design process and then shift to tools that produce interactive and functional interfaces later. Overall, as a team moves forward in their design process, they typically move from lower-fidelity representations of their design toward more dynamic prototypes and functional code that can be tested with users and verified for functionality.

When to use tools also depends on the maturity of the product being designed and the skills of the designers working on the team. Products that are undergoing small iterations, such as a minor update to a mobile application, may see design teams working directly with a high-fidelity interface prototyper to make design changes and rapidly test the new interactivity. Teams with a mature set of pre-built components used across the entire product may also work in an interactive prototyper to build new pages for an application since they are focused more on the composition of known elements. The team may likely use a drawing tool to map out a new user flow and jump straight into the interactive prototype or even an IDE to put the elements together into a functional interface. On the other hand, teams working on new product development who do not already have a mature design system may focus more time working with drawing tools to figure out how a design will be structured and how various functional components will look. Finally, some teams with designer–developers may prefer to work directly in code to lay out and prototype new designs. Such a process can allow a team to be fully grounded in technical capabilities and can help to reduce product development time. Such straight-to-code processes are often supported with robust design systems where the code for design elements is already established, such as in web-based application development. Overall, UX design tools can be mixed, matched, and used fluidly throughout a design process.

The benefits of using computerized UX design tools include:

(1) Designs can be rapidly prototyped and implemented.

(2) The lower-level widgets used in a UI from a toolkit and other libraries as well as other elements in a design system will be more usable. Since these are used across many UIs, more effort can be expended on them than may be practical on any single-user interface since they will be used with many different applications. They are also less likely to have bugs in their implementations and are more likely to be accessible to people with disabilities.

(3) Different applications are more likely to have consistent user interfaces if they are created using the same set of user interface tools, such as the same platform toolkit or the same design system.

(4) It will be easier for various specialists to be involved in designing the user interface rather than having the user interface created entirely by programmers. Graphic artists, UX specialists, and technical writers can all be involved, even when they are not able to code.

(5) A design can be experienced interactively in ways close to the final implementation, which can support more accurate testing of a design concept (in contrast to using static or paper-based tools).

(6) Tools can facilitate the design prototypes to be documented, annotated, and version-controlled digitally. Working with digital tools allows the digital files to include additional metadata useful for the overall design work, such as notes about functionality or specification of visual properties. Designers may leverage commenting systems to deliberate on design decisions and assign design tasks to one another. And finally, digital design files can be put under version control, allowing for backups and easier tracking of design changes over time.

(7) Multiple people can more easily work on the same design prototype at the same time in a remote and distributed manner. Today, most UX design tools are being built with web-based communication, allowing multiple users to work concurrently on the design file. This is increasingly important for teams where multiple designers are working on the same product and when teams are remotely distributed.

20.5 Toolkits and Frameworks

As defined above, toolkits are a library of widgets, whereas frameworks provide an OO structure that defines the architecture of an application. In practice, this distinction is somewhat fuzzy, and developers argue about whether various tools "count" as a framework. For example, I would classify ReactJS as a framework

since it provides `React.Component` and other classes from which developers should inherit, but their website (reactjs.org) calls it a "library" and differentiates it from other systems that it labels as "frameworks."

20.5.1 Look and Feel

As mentioned in Section 2.5.2.1, DOS and other early systems did not try to enforce a consistent look and feel, and they assumed that vendors using their platforms would use various widgets and toolkits. The Macintosh (from 1984) recognized the importance of UI consistency and the ability of the toolkit to support it, so all Macintosh applications were expected to use its toolkit, called the Macintosh Toolbox, and to conform to the Macintosh Style Guidelines. The Toolbox made the "path of least resistance" [Myers et al. 2000a] (what was easiest for the developers) be to follow these guidelines, which was a key factor in the consistency of the UIs of early applications.

Interestingly, when the X window manager was being developed around the same time by a consortium funded by workstation companies (see Section 11.3), the designers explicitly rejected the idea of requiring adherence to a specified UI since the companies at first could not agree on a common look and feel. The mantra of the X window manager and its initial toolkit, called Xt (from about 1988) [McCormack and Asente 1988], was "mechanism not policy." They said:

> "Instead of mandating a particular user interface, the base window system should support a wide range of interfaces. To achieve this, the system must provide hooks (mechanism) rather than religion (policy)." [Scheifler and Gettys 1986, p. 83]

Therefore, Xt focused on what they called the *intrinsics* of how the widgets would be implemented, expecting that one or more separate look-and-feel layers would be built on top.

As described in Section 2.4.12, the consortium discovered that this approach did not provide the desired portability, and applications written using Xt had to be recoded for each vendor's computer system and each look and feel. Therefore, they formed two rival camps supporting two different look and feels, OpenLook and Motif, and eventually coalesced around Motif as the X toolkit for applications.

20.5.2 Frameworks

Apple quickly discovered that developers were having a difficult time using the Macintosh Toolbox, so they created what arguably could be called the first user interface *framework*, MacApp, in 1986 [Schmucker 1986]. The original Macintosh operating system and Toolbox (see Section 28.2) were implemented in assembly

language and Pascal (a programming language created around 1970 by Niklaus Wirth, designed to be easy to learn and use), but for MacApp they recognized it would be easier if it was implemented using an OO language (as shown by Smalltalk from 1980, see Sections 2.4.3.1 and 20.12.7). Therefore, Apple hired Niklaus Wirth as a consultant to build *Object Pascal* as a new programming language in which they provided classes for Window, Command, Document, and many other elements (described in Section 20.12), which the programmer would subclass. Subsequent UI frameworks have used a quite similar structure, implemented in most popular programming languages, notably Objective-C (which supplanted Object Pascal for Apple development starting when it absorbed NeXT about 1996), C++, Java, and JavaScript.

20.5.3 Toolkit and Framework Class Hierarchies

In the initial toolkit libraries, built with non-OO programming languages such as Pascal and C, the widgets were provided as a set of routines that a programmer was required to call in a particular order. For example, in an early version of the Macintosh Toolbox, the programmer would call `GetNewMBar` to create a menu, and then `SetMenuBar` to fill its items, followed by `DrawMenuBar` to actually display it.

In the next phase, toolkit creators, recognizing that it was easier to implement user interfaces using OO techniques—as demonstrated by Smalltalk (Section 2.4.3.1)—built hacky special-purpose OO packages on top of C. Examples include the intrinsics layer of Xt (Section 20.5.1) and the lower layers of the Andrew Toolkit [Palay et al. 1988]. One reason for this was that AT&T, where Bjarne Stroustrup had developed C++ from around 1982, was not willing to let the community use C++ for free at first, and also C++ was missing important features that developers felt were important for UI development, such as dynamic linking of code [Palay et al. 1988]. Eventually, these issues were overcome [Linton et al. 1989], and the world standardized on using C++ and its successors like Java and JavaScript (except for Apple, which continued to use Objective C).

With this rise in popularity of OO languages for implementing UIs in general, and toolkits in particular, it was recognized that many widgets share features, and that OO inheritance could be used to share the code for those features. For example, the menu that pops up for a combo box should look and feel like the menu that pulls down from a menu bar (see Section 7.2). Therefore, OO toolkits developed elaborate and deep inheritance hierarchies to share code. For example, the class hierarchy for package `javax.swing` in Java 10, which provides all the UI features for Java, contains 6,002 classes and interfaces, with an inheritance depth of at least 8. A developer who wants to use a widget must know which of these classes to create instances of and how to coordinate multiple classes together, which can

be very difficult to figure out [Ko et al. 2004]. For example, there are 68 different classes and interfaces with the word "menu" as part of the name in Java 10, and the one with the simplest name (like the *two* different classes both named "`Menu`") is often *not* the appropriate one to use. This complexity has arisen from the desire for these toolkits to support a wide variety of widgets and also, as discussed in the previous chapters, the subtle differences in the ways that widgets should work. For example, the menu for a combo box does *not* work exactly like the menu that is part of a menu bar (consider the differences in the default selection or what happens when the user aborts), so there turns out to be less sharing and more specific classes than might be expected.

20.5.4 Research Toolkits and Frameworks

Although most research toolkits have focused on helping developers to *use* a collection of widgets [Palay et al. 1988, Pausch et al. 1992, Ousterhout 1994, Roseman and Greenberg 1996, Salber et al. 1999], including controlling their properties and laying them out, a few have also provided mechanisms designed to help developers *create* new interaction techniques. For example, my Garnet (1988) [Myers et al. 1990] and Amulet (1994) [Myers et al. 1997a] toolkits provided a novel low-level input handling model called *interactors* (described in Section 20.12.4) that replaced using event handlers, and they also provided *constraints* as a way to both pass values around inside widgets and handle layouts of parts of widgets and the widgets themselves (Section 20.6.2). Garnet and Amulet also provided built-in support for investigating gestural interactions, as discussed in Section 14.4. The InterViews research toolkit (1989) was one of the first to show that C++ could be effectively used to create toolkits [Linton et al. 1989], and it provided an innovative way to support text editing interaction techniques [Calder and Linton 1990]. The subArctic toolkit (1996) provided an efficient implementation of constraints [Hudson and Smith 1996] and a highly influential input handling mechanism [Hudson et al. 2005], which were useful for implementing novel interaction techniques.

20.6 Geometry and Layout Management

There have been many approaches to how programmers specify *where* the widgets should be placed and how *big* they should be. In some of these approaches, the programmer sets the location and size into the widget itself, and in other cases, they are defined by the *container* of the widget. This section discusses popular options.

20.6.1 Absolute Layout

The simplest way for a programmer to specify where a widget is placed, for example, inside a window or dialog box, is to just pick an explicit X/Y coordinate, either by

putting those constant values in the code or resource file (explained in Section 20.7) or by using an interactive widget layout tool (discussed in Section 20.8). For example, a ribbon menu might be specified to start at location X=2 and Y=50 within a window (the coordinate system for all window graphics start with 0,0 at the *top*, *left* corner, with X increasing to the right and Y increasing down).

However, absolute positioning is often not sufficient. For example, the *width* of the ribbon menu should depend on the width of the window it is in, and menu items in the ribbon might move based on the window size, even jumping to a second line or disappearing into a submenu if the window, and hence the ribbon, is too narrow. Similarly, the size of a menu must usually be calculated based on the size of the menu's items, which may even be determined at run-time, not design time.

20.6.2 General Constraints

The second simplest (conceptually, not necessarily for a programmer) way to specify the layout is to use *constraints*. A *constraint* is a relationship among objects that is declared by the programmer and then maintained by the system. Examples of constraints include that the width of the ribbon menu should stay the same as the width of the window minus 4 pixels (e.g., if the window border is 2 pixels wide on each side). Another example is shown in Figure 10.11(c), where one end point of the line is constrained to stay connected to the point of the triangle, even if the triangle moves or changes size. Once the programmer has set up the constraints, it is the system's job to determine values that *satisfy* the constraints.

As mentioned in Section 2.4.1, this kind of general-purpose constraint was first demonstrated in the SketchPad system from 1963 and has been further developed since then by a large number of toolkits, both research [Borning and Duisberg 1986, Szekely and Myers 1988, Freeman-Benson 1990, Myers 1990c, Zanden et al. 1991, Gleicher 1993, Hudson 1993, Sannella 1994, Oney et al. 2012] and commercial (e.g., Apple macOS and iOS). These systems have used a wide variety of ways for programmers to specify the constraints, including regular code, mathematical notation, and special-purpose domain-specific languages (DSLs), along with a variety of algorithms for solving the collection of constraints to determine values for each object.

Note that in many systems the constraints can be used for purposes beyond layout. For example, the Garnet toolkit [Myers et al. 1990] was one of the first to show that constraints are useful for passing data values around the user interface—for example, the Boolean value that controls whether the Delete menu item is invalid (grayed out) can use a constraint depending on whether a graphical object

is selected or not. Similarly, when using the MVC pattern (Section 20.12.6), constraints can be used to keep the model value and the view values in sync. This kind of constraint—to keep values in different parts of the code synchronized—is now called *data bindings* and is supplied as a separate feature in some UI toolkits, such as Angular and Flex for Flash (starting around 2004) from Macromedia/Adobe/Apache.

However, general-purpose constraints and data bindings have a number of disadvantages [Myers et al. 2000a]. First, programmers have found them difficult to author and debug when set up incorrectly. When there are too many or conflicting constraints, there may be no solution, and when there are too few, there may be multiple or even infinite solutions, and the system is likely to pick one that the programmer does not want. For example, a classic layout problem is that many times all constraints can be solved by just making all the positions and sizes be 0, which is never what the programmer wants. Determining the cause of these problems is often difficult. Second, a related problem is that constraint authoring mechanisms often have proven to be difficult to learn, so they are hard to use by beginners. This is due to the special-purpose languages or APIs for specifying the constraints and the need to think about the specification of the problem in a new way (e.g., as declarative constraints rather than imperative actions). Third, these kinds of constraints can be computationally expensive since the system often must *plan* how to solve the constraints in addition to actually solving them, and many more computations are usually required for the solving step since the system often cannot determine exactly what needs to be re-evaluated. This was a key motivator for the redesign of the AngularJS toolkit into Angular around 2016—to match the performance of React and other competing toolkits.

20.6.3 Layout Managers

Due to the complexities just mentioned for general-purpose constraints and data bindings, another approach is to use special-purpose *layout managers*. There are three key kinds.

20.6.3.1 Special Values for Coordinates

The simplest way to provide more flexible layout compared to absolute values for the position and size is to generalize the values that can be used. For example, HTML and CSS have long had the option of having an image “float left” or “float right,” which means that it will stay at the edge of the current container (often the document or table cell) even if it changes size. Similarly, values can often be

computed based on their container, as in having an object be centered. The size can often be computed based on the container, such as using "75%" instead of a specific size.

20.6.3.2 Special Kinds of Containers

The most popular way to provide layout is to use special kinds of containers. For example, a *grid* container would line up all of its contents into a two-dimensional (2D) layout of rows and columns. The obvious example is an HTML *table*, but there are grid containers in many toolkits, such as the Java and Android SDKs and Tk [Ousterhout 1994]. Often, as with HTML tables, an entry can span more than one row or column, but this is still a fairly restrictive layout. Other common options are to flow with wrap-around, which is what happens with text in paragraphs and with some horizontal menus, or to be in a vertical list with scrolling as in many kinds of menus such as for fonts.

However, these layouts are insufficient for many common layouts in UIs. For example, a scrollbar should stay at the right edge of a document, with a fixed width but variable height. The insides of the scrollbar should have buttons of constant size at the ends, with a variable sized middle section (the "trough," see Section 5.6). Similarly, the menu bar should stay at the top of a window and change width proportionally to the document. Toolkits often provide more controllable containers to support these common layouts, such as the Motif "Form" and Java "GridBagLayout," but these are notoriously difficult to use, as admitted by the official documentation (Form: "For details, see the Motif Reference Manual, because the complete behavior of Form is quite complicated.") and shown by a funny video for GridBag from around 2003.[2]

Another complexity with these special containers is that the layout and sizing must often be performed in multiple passes. For example, a vertical menu might center the items, which requires knowing the size of the container, but that requires knowing the width of the widest item. Some layout managers allow items to have minimum, maximum, and desired sizes, so an item may tell the container its desired size but then be told by the container to actually use a different size.

However, despite these complexities, special containers are still the most common option for layouts, beyond using fixed coordinates. For example, the current version for HTML/CSS/JavaScript is called a *FlexBox*.

2. The video "Totally Gridbag" was authored by Matt Quail in Flash in July 2004, which no longer runs (http://madbean.com/anim/totallygridbag/). Fortunately, there is a YouTube version: https://www.youtube.com/watch?v=UuLaxbFKAcc.

20.6.3.3 Stretchy Spacers

The final common layout mechanism uses special *spacer* items interspersed among the displayed items, where the spacers may have fixed or variable sizes and may be stretchy. The first use of this model was by Donald Knuth in his original TeX formatting system from 1978, where the stretchiness of spaces could be controlled to adjust how lines were justified. As mentioned in Section 9.9, this was adapted by the InterViews toolkit [Linton et al. 1989] as its *boxes and glue* metaphor to provide a different way to specify the layout of widgets and other UI elements. The programmer includes glue objects and sets their original size, shrinkability, and stretchability inside layout boxes to achieve the desired layouts (see Figure 20.2).

A similar approach used an analogy of *springs and struts* to represent the stretchiness of the spacer objects. Struts had fixed length and no stretchiness, whereas springs were stretchy to various degrees. This form was used by the NeXT UI toolkit (Section 2.4.11) and also by a cross-machine framework called *Galaxy* from Visix Software [Visix Software Inc 1997].

20.7 Resource Files and Internationalization

Another question about the properties of widgets is where to define them. The most obvious answer is in the code—so a programmer creating a button might provide the position and size and string label as constant values in the code (or, if defined by a layout manager, those values might still be saved in a code file). However, when

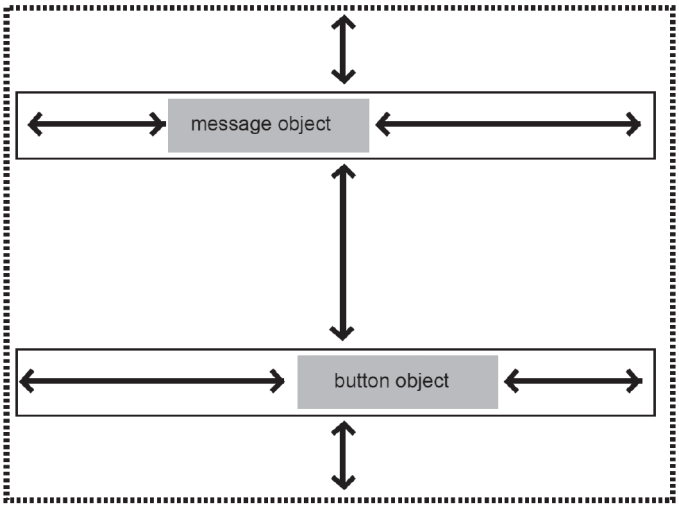


Figure 20.2 The stretchiness of the glue objects in InterViews, represented by double-headed arrows, controls the horizontal and vertical placement of the message and button objects. Source: Fig. 4 of Linton et al. [1989].

the user interface is translated into a different foreign language (like from English to French), the code itself will need to be edited, and the programmer will have to laboriously go through to find all the values that need changing. Note that it is not sufficient to just change the string values for the labels since they will be different sizes in different languages, requiring the positions and sizes of widgets to change as well (see Figure 20.3).

This gave rise to *resource files* in the original Macintosh (Section 2.4.7), where all the labels, positions, and sizes were in the separate *resource fork* and the code referred to the values using abstract identifiers. Thus, the same code could be used without recompiling in different languages just by swapping resource files. This also made it easier for professional translators to be involved since they could modify the resource files without needing to be able to program. Bruce Horn (Chapter 29) invented the resource file for the Macintosh also to save space in the code since the resource data could more easily be read into memory from disk and discarded when that part of the UI was not needed. Another advantage was that interactive editors used to define and modify the user interface (Section 20.8) could

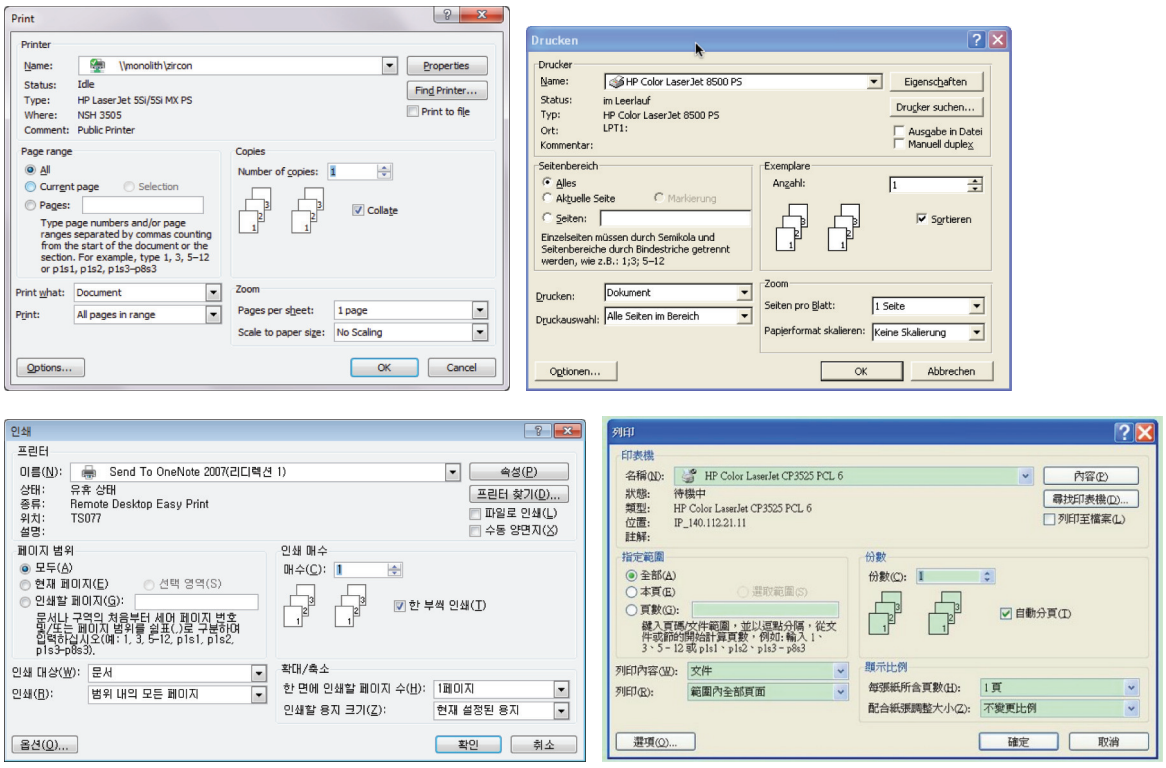


Figure 20.3 The "Print" dialog box in Microsoft Word in different languages. Sources: Screenshots of Microsoft products, used with permission from Microsoft.

read and store their values in the resource file without necessarily having to modify the application's code. In this case, the resource files would include many other parameters for widgets and dialogs, including colors, styles, fonts, icons, pictures, sounds, the callback functions to call when the widget was operated, and so on.

The older systems defined their own syntax for the contents of the resource files, which evolved to be fairly sophisticated with wildcards in systems such as Motif (Section 2.4.12). With the advent of XML as a standard encoding format, many systems switched to using XML, such as the XAML language from Microsoft, in use since 2008 to define the properties for the .NET Framework widgets (and more).

One disadvantage of using resource files is that it can make it much more difficult for programmers to understand or search through the code since the identifiers in the code are often not as meaningful as the labels on widgets would be. Further, if the programmer sees a widget in the user interface, that doesn't necessarily help the programmer find that widget in the code since the string shown on the widget never appears anywhere in the code, so the programmer would first need to find and search the correct resource file to find the associated ID. There are also the problems of user interfaces for right-to-left natural languages like Hebrew and Arabic, as well as the need in some languages for entirely new widgets or even whole new tabs or dialogs (see Figure 20.4).

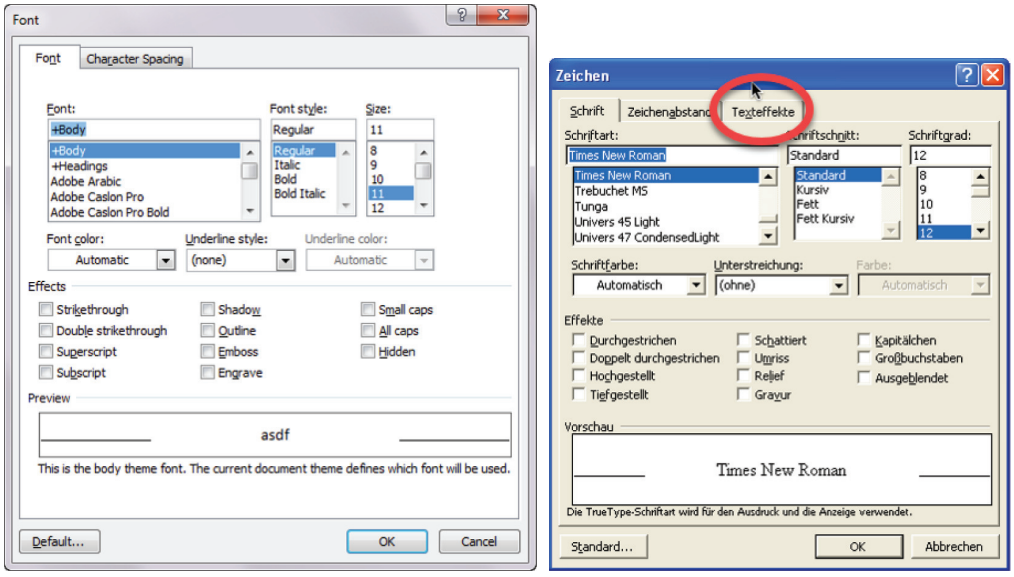


Figure 20.4 The "Font" dialog box in Microsoft Word in different languages. Note the extra tab in the German one (circled at right). Sources: Screenshots of Microsoft products, used with permission from Microsoft.

20.8 Interactive and Graphical Tools for Building UIs

As summarized in Section 2.5.2.2, using interactive tools to define the layout and properties of widgets has a long history in research and commercial products, dating back to at least 1983 [Buxton et al. 1983b, Stevens et al. 1983]. Possibly the first commercial example was the ResEdit tool for the original Macintosh (1984) for editing the resource files where the properties of the widgets were stored. Many other commercial and research systems were subsequently created, calling themselves *Resource Editors*, *Interface Builders* (IB), *GUI Builders*, *Interface Development Tools* (IDT), or *Form Editors*. In general, they all have a menu (palette) of the kinds of widgets that can be created, a main window where the user places those widgets, and property sheets or dialog boxes for defining the properties of each widget (see Figure 20.5).

One story is that Jean-Marie Hullot, who created the NeXT Interface Builder, previously created the SOS Interface in Lisp for the Macintosh while working at INRIA (1984). This allowed widgets and their layout and properties to be specified interactively. He told me in an email from August 20, 1996:

> Yes I did create a product called "SOS Interface" when I was working at INRIA. The project was funded by the French government. It started in late 1984 and was finished early 1986. The first public presentation was at the Apple University Consortium conference in Cambridge (UK) in March 1986. It was built on top of the "Le Lisp" system also developed at INRIA and was working on Macintosh. ... [T]he motivation behind this project was to get a tool allowing my students to program on the Mac without having to learn the toolbox which was not part of the curriculum. Maybe my main source of inspiration has been the Resource Editor that was part of the Macintosh development system. As far as building the UI with a drawing-like program, [ResEdit] was pretty good, although a little intimidating for many. What was really missing and which was the key innovation in SOS Interface was the binding mechanism between UI widgets and source code. Later in 1986 I moved out of INRIA and rewrote a similar tool, still on a Lisp system on the Mac (ExpertLisp), but this time it was a real commercial product instead of being a research project. I called it "Interface Builder" and it was commercialized in the US by ExperTelligence, a company based in Santa Barbara in 1987. To my knowledge this was the first such tool on the market. In 1987 I met Steve Jobs who got really excited about Interface Builder, I sold it to NeXT and joined NeXT, where I did make a full rewrite with tons of improvements and new things in Objective C. At the same time I did most of the NeXTSTEP UI design with Keith Ohlfs (graphic artist) and Steve himself, and

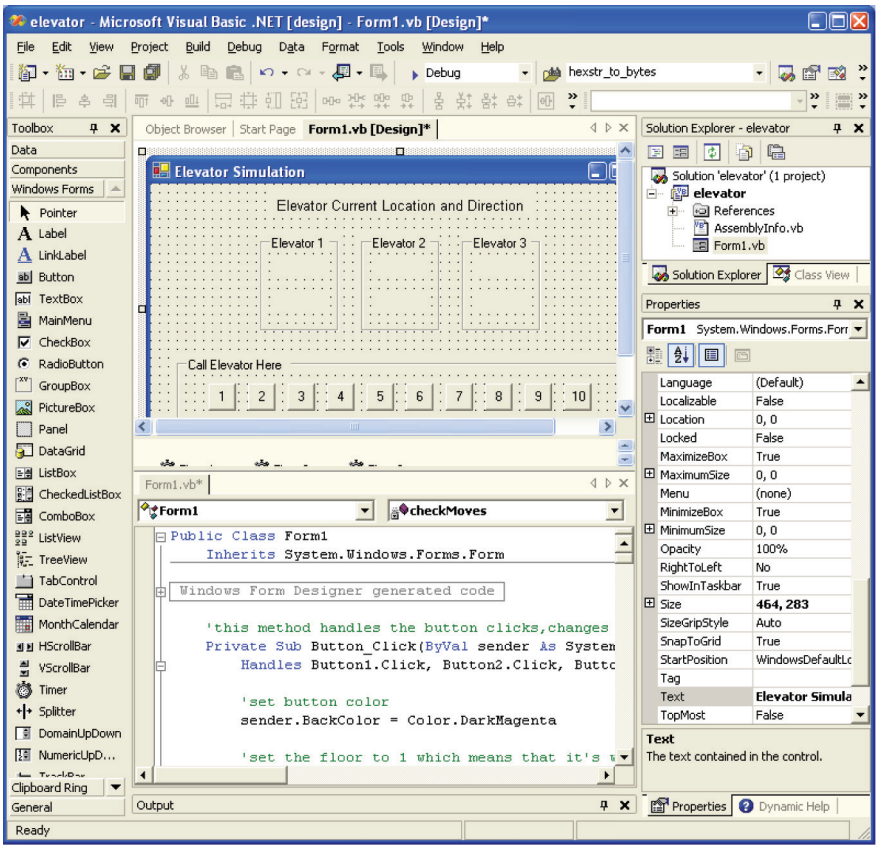


Figure 20.5 Screenshot of the Visual Basic system, with the palette of widgets on the far left, the form editor at the upper center where the user is laying out widgets, the property sheet for the selected widget (here, the whole Elevator Simulation dialog box is selected), and the code editor at the center bottom. Source: Screenshot of Visual Basic System.

did participate heavily do the design of the NeXTSTEP toolkit. That's all the story. I hope it will help.

Another important commercial example is Microsoft's *Visual Basic* (VB), which combined a GUI Builder tool (called its *Form Editor*) with a version of the Basic programming language, first released for Windows 3.0 in 1991. The goal was to make it possible for more people (ideally "everyone") to create applications on Windows by selecting and placing widgets and higher-level pre-build components and then writing small amounts of "scripting" code to connect them together. We now call this goal *end-user programming* (EUP) [Myers et al. 1992] or *end-user development* (EUD) [Lieberman et al. 2006], and it has proven more difficult than expected.

Visual Basic added features to both the form editor and the programming language that made them both more complex, essentially acknowledging that they require people to have a fair amount of programming expertise, culminating in a radical change in 2002 when the tool was converted to use the C# OO language, and the Visual Basic programming language was converted to being OO as well (this version is shown in Figure 20.5).

Some tools, especially from research, provided extra capabilities beyond placing widgets at fixed locations and setting properties. For example, a number of tools provided a GUI for specifying constraints or layout managers for the placement of widgets (Section 20.6). The OPUS editor [Hudson and Mohamed 1990] allowed users to interactively place reference lines and constraints among the widgets and

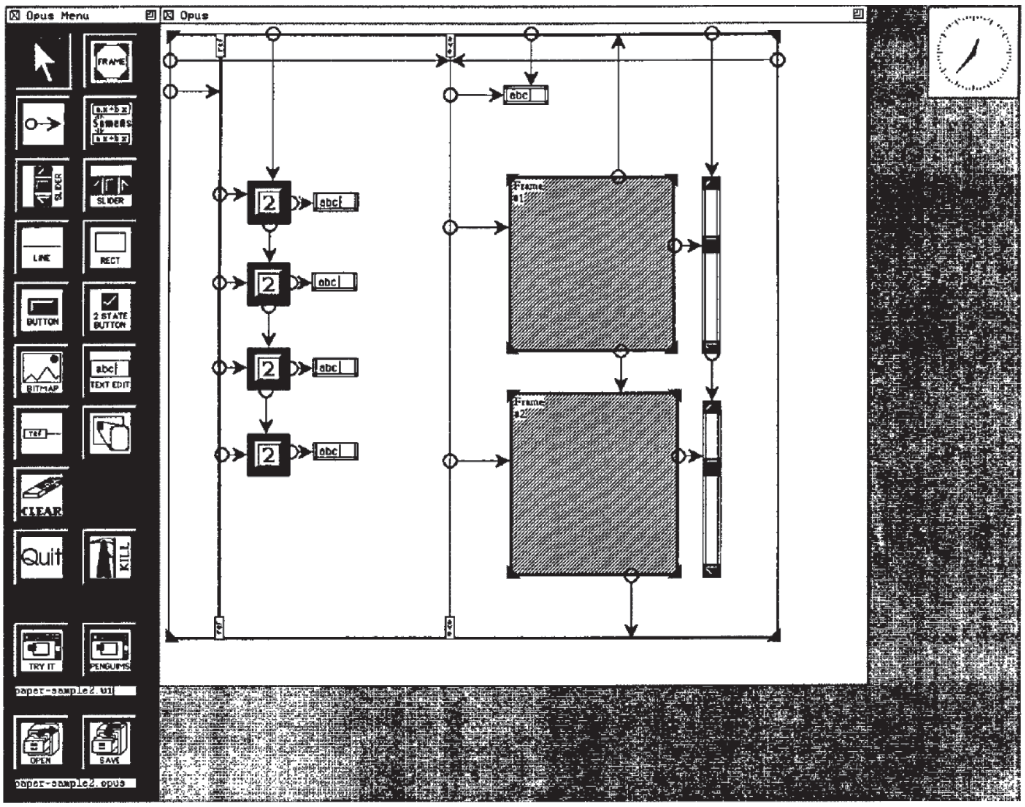
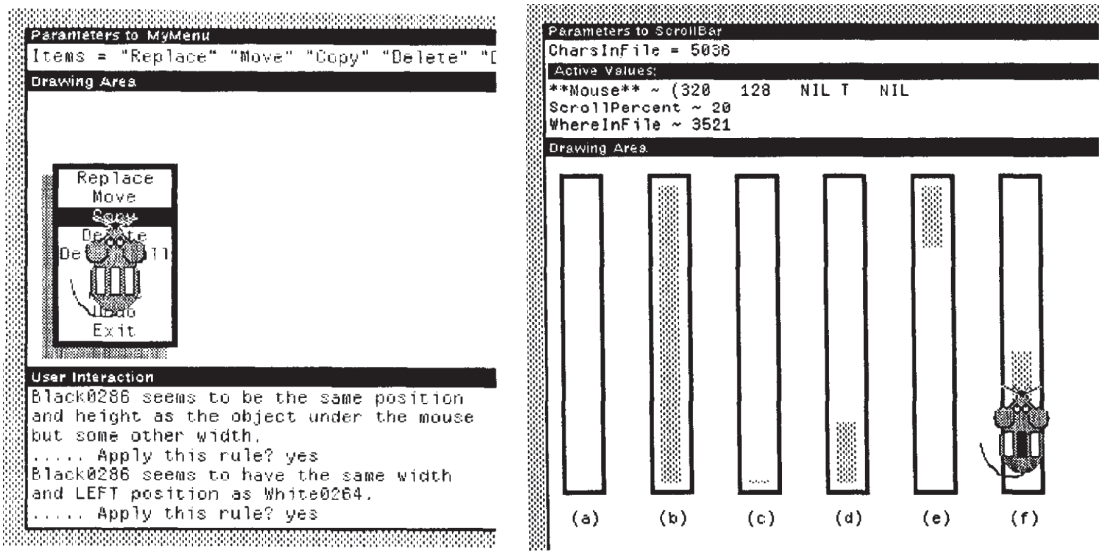


Figure 20.6 The OPUS editor allows the user to place *constraints* as lines with arrowheads among the widgets and references lines (without arrowheads). Source: Fig. 1 of Hudson and Mohamed [1990].

the window so that the layout would adjust appropriately as the size of the window changed (Figure 20.6). Other interactive tools have allowed springs and struts (Section 20.6.3.3) to be placed interactively.

Another set of tools, mostly from research, strived to allow new widgets to be *designed and created* interactively. One of the earliest of these was my *Peridot* tool from 1986 [Myers and Buxton 1986], which allowed users to draw rectangles, text, and other graphic elements to define the look of a new menu, scrollbar, or other widget, and then to demonstrate how it changed in response to mouse events (see Figures 18.9 and 20.7). Most subsequent research *programming by example* (PBE) systems also focused on defining the look and behaviors of interaction techniques (see summaries in the two PBE books [Cypher et al. 1993, Lieberman 2001]). Another approach was to create a special-purpose DSL for describing the behaviors, such as the combined spreadsheet and transition diagram (Section 19.4) language used in InterState [Oney et al. 2014] (see Figure 20.8).

Of course, regular HTML/CSS can define different looks for an element when idle or hovered and also provide an action on click, so users can easily create their



(a) (b)

Figure 20.7 The user can show the look and behavior of widgets by example with Peridot. (a) Starting from the menu shown in Figure 18.9, the user can demonstrate how the highlight moves with the mouse. (b) Similarly, the various stages of drawing what a scroll bar looks like with rectangles, followed by demonstrating how the gray indicator bar should move with the mouse's middle button. Sources: (a) Fig. 3f of Myers and Buxton [1986]. (b) Fig. 2 of Myers [1987a].

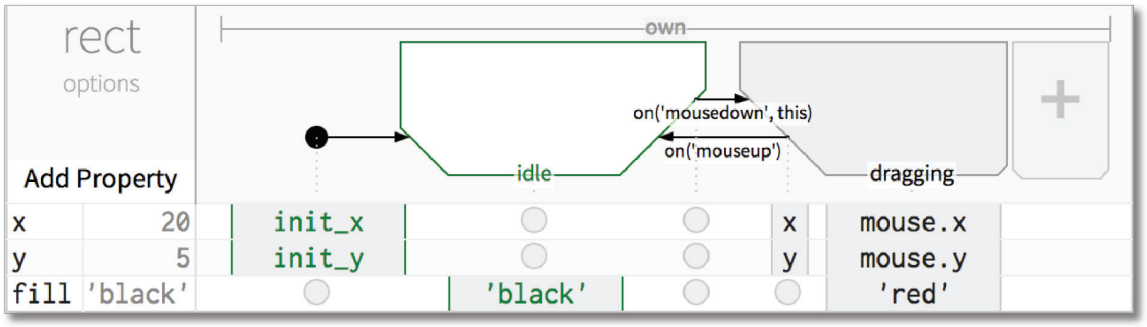


Figure 20.8 InterState's transition diagram with two states: idle and dragging, with transitions due to mousedown or mouseup. The current state controls which spreadsheet columns are in effect for the properties at left. Here, the X,Y properties are using the initial values since it is idle, but if it switches to dragging, then the mouse.x and .y will be used instead. Source: Fig. 1 of Oney et al. [2014].

own menus and buttons. There has been at least one commercial tool that focused on helping users define more varied behaviors for new widgets interactively—with Adobe Flash Catalyst 5.5 from 2011, the user could draw graphical elements in Flash and then select a behavior from a menu (Figure 20.9).

20.9 Interactive Prototyping Tools

In addition to the interactive tools for *creating* parts of the user interface discussed in the previous section, there have also been many tools to help *prototype* the interface, including exploring both how to use existing widgets and how new widgets should look and behave. As defined above (Section 20.3), the key distinction is that with prototyping tools only a *simulation* of the UI can be made, which cannot be used as the real code of the application. These tools have also been called *prototypers*, *wireframing tools*, or *click-through prototypes*. Of relevance to this book, these tools can be used to both prototype interfaces using widgets and to prototype new designs for widgets themselves.

20.9.1 Kinds of Prototyping Tools

One distinction made about prototyping tools is the level of fidelity. *High-fidelity* (hi-fi) prototypes often look very much like a real user interface, and some of the elements may appear to work, but there is usually no back-end programming. For example, a prototype of a font selection menu may look realistic, but the same font would be returned no matter what is clicked. In contrast, *low-fidelity* (lo-fi) prototypes strive *not* to look like the real UI. Some low-fidelity prototypes are drawn with

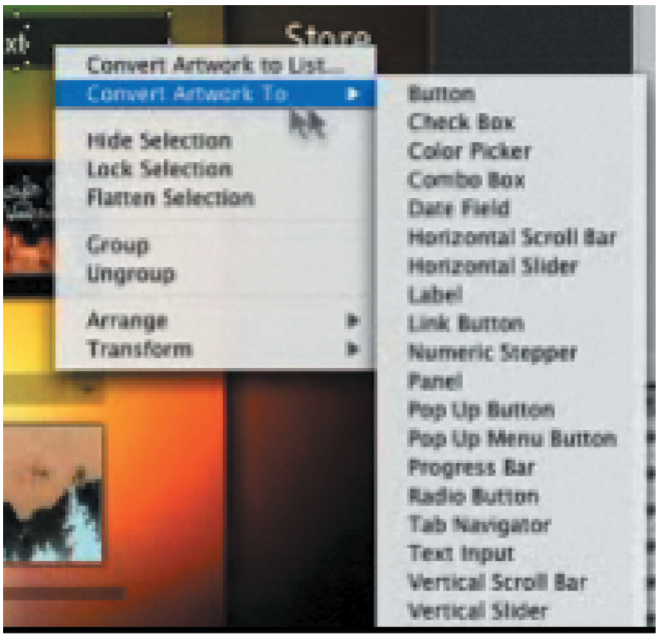


Figure 20.9 The behavior menu from Adobe® Flash® Catalyst 5.5. Source: Adobe product screenshot reprinted with permission from Adobe.

pens and pencils on paper (called *paper prototypes*), or they may be drawn with special prototyping tools that have a sketchy look. Often low-fidelity prototypes[3] are called *wireframes* since the edges of widgets and other elements are represented as lines (as if they were framed by wires), as shown in Figure 20.10. The elements included in a prototype (no matter whether sketchy, wireframe, or hi-fi) may allow the user to click on them to transition to a different page, in which case it may be called a *click-through* prototype. In other cases, the experimenter "plays computer" and switches from one page to another to pretend that the interface actually works. Note that these distinctions do not necessarily constrain which kind of *tool* the designer uses—often the *same tool* can create low- or high-fidelity prototypes, and the output of a tool may support click-through or else require the designer to manually switch pages.

20.9.2 Examples of Prototyping Tools

One of the earliest prototyping tools was Dan Bricklin's DEMO (1985) (Chapter 26), which created prototypes of DOS or other character-display interfaces (explained in

3. Some authors (e.g., Buxton [2007]) make a distinction between sketches and low-fidelity prototypes, but I use the latter to include the former in this book.

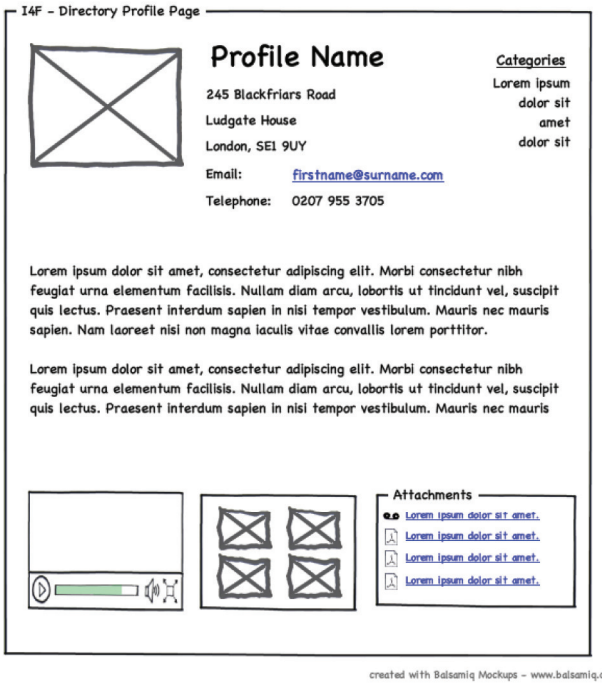


Figure 20.10 Example of a wireframe prototype. Source: Doos, https://www.flickr.com/photos/doos/3931846833/, CC BY 2.0, https://commons.wikimedia.org/w/index.php?curid=12888965.

Section 2.3). The designer could make pages of the UI by moving the cursor to any part of the screen and typing what should appear there. Areas could be defined as input fields or text that was editable by the end-user, as well as buttons and menus that could transfer to different screens (so it created what we now call *click-through prototypes*). Since the prototype looked like the real application screens, DEMO would count as a high-fidelity prototyper.

HyperCard (1987) (Section 2.4.9) was intended to provide the ability for the general public to create real applications, but it had little success in this goal. However, it was enormously influential and useful as a prototyping tool. A key factor was that, unlike previous drawing and UI tools, new pages were loaded *into the same window*. Thus, the user could draw before and after pictures of what an interaction looked like to easily create click-through prototypes. There was also an easy-to-learn scripting language called HyperTalk that allowed more sophisticated behaviors to be created. Macromedia's Director product, also from 1987, was originally intended for creating multimedia applications and movies, but since it had some support

for different actions based on input events, it also was used as a prototyping tool, especially for more elaborate *future scenarios* [Myers 1992a]. Both HyperCard and Director were primarily used for hi-fi prototypes.

Subsequently, there have been many successful prototyping tools, usually targeted at creating click-through prototypes at various levels of fidelity. Examples include Axure (from 2003), Balsamiq (2008), Figma (2016), Adobe XD (2017), and many others. All of these provide templates so that the prototype can *look* like various web or phone screens, the ability to draw wireframes or import elaborate graphics, and support for clicking on regions to go to different screens. Other important functions include components or modules so portions of the UI can be designed as a unit and included as a part of multiple screens, and features to help with user testing. Some popular drawing programs like Microsoft PowerPoint, Adobe Photoshop, and Illustrator have also added rudimentary click-through (hyperlink) behaviors, so they can be used for prototyping directly. Finally, many people report finding it easy enough to create web pages so that these are used as prototypes, even for UIs that will not be in a browser.

20.9.3 Research on Prototyping Tools

As mentioned in Section 2.5.2.2, early research tools for interactive UI layout did not distinguish whether they were aiming for building prototypes or real user interfaces (and thus were aiming to be interface builders). Early examples that supported layout of click-through UIs include Trillium (1981) [Henderson Jr. 1986], Steamer (1979–1985) [Stevens et al. 1983], and the MenuLay system (1983) [Buxton et al. 1983b].

More interesting may be research tools that focused on the *sketching* phase of prototyping, which is recommended to happen before wireframing [Buxton 2007]. Our SILK tool (1995) [Landay and Myers 1995], created by my former Ph.D. student James Landay, was discussed above in the context of gesture interfaces (Section 14.3.3). SILK allowed designers to draw user interfaces by sketching with a stylus, and the designs stayed sketchy even though the system would recognize which widgets were intended (see Figure 14.2). SILK could convert widgets into a version that looked more high fidelity, and another feature supported storyboards to show which screens should be transitioned to when clicking on widgets (see Figure 20.11) [Landay and Myers 1996]. A similar storyboard feature is now provided by many commercial tools. When James Landay became a faculty member, he further investigated sketching interfaces with his students, such as the DENIM system, which supported sketching of web user interfaces at varying levels of resolution [Lin et al. 2000], and Damask, for sketching user interfaces targeted at multiple devices [Lin and Landay 2008].

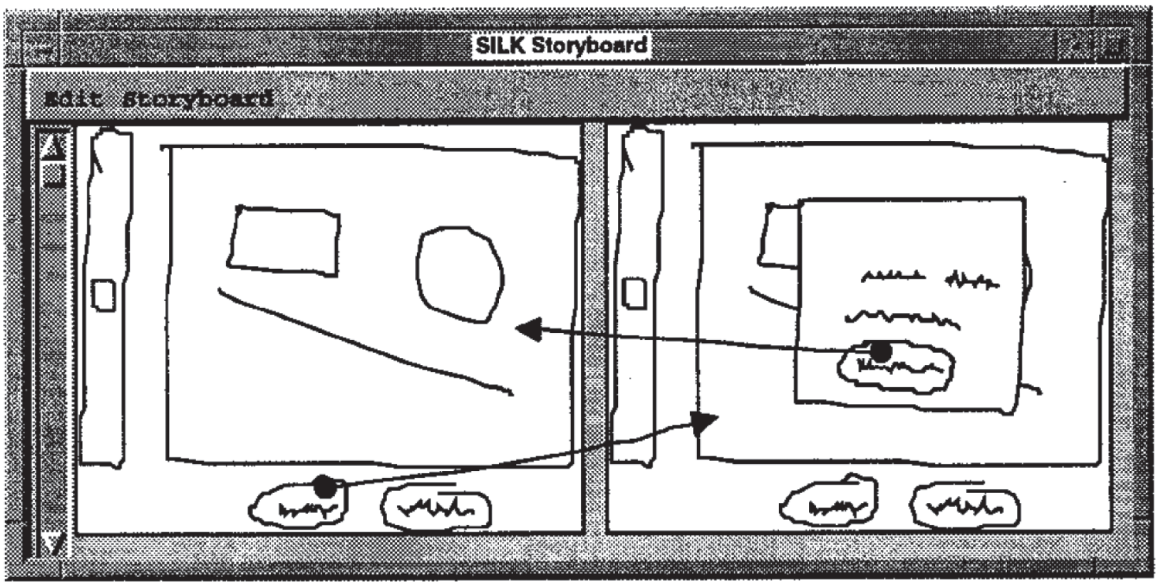


Figure 20.11 Storyboard in SILK shows transitions for buttons (left) that pop up a dialog box and (right) that make it go away. Source: Fig. 2 of Landay and Myers [1996].

20.10 Automatic Creation from Models

A completely different way to create user interfaces out of widgets is to provide a high-level description of what is wanted (sometimes called the *model*) and have a system automatically create an interface. A key motivation is to adapt automatically to different devices or users with different properties. A simple example of this is the way that HTML will determine the layout of a collection of input widgets based on their size and the width of the window. However, most systems in this category try to automatically decide on more than just layout. For example, many will automatically select *which kind of widget to use* based on various properties such as how many choices there are and how much space is available. A model-based system might choose radio buttons if there are a small number of choices and sufficient room, or else a popup combo box if there are between 5 and 100 choices, or else a text field with autocomplete if there are more than 100 choices.

This was a very active research area in the 1990s through the early 2000s, but it has not had a significant impact on commercial practice for UIs. I think this is because (1) the automatically produced UIs were just OK, not wonderful; (2) designers want to have more control over the UIs; (3) existing mechanisms that let designers specify layouts for two or three sizes (e.g., desktop, tablet, or phone layouts) are sufficient; and (4) with some of these automatic systems, the programmer

must provide a specification that is longer and/or harder to author than just writing the code—for example, the Mastermind system [Neches et al. 1993] envisioned the designer creating models of the users, the target displays, the application features, and the tasks that users would do, which was way too much work in practice (see also Myers et al. [2000a]).

Nevertheless, here is a quick summary of some important model-based systems [Sukaviriya et al. 1994]. The original Macintosh had very specific rules about how its UIs should look, and the Mickey system [Olsen Jr. 1989] embodied these in a tool that took a Pascal program with special comments and produced the UI. It could handle the main menus and some simple dialog boxes. The Jade system from my lab [Zanden and Myers 1990] required a specification in the Lisp programming language and could produce output with multiple look and feels from the same specification—for example, to match Motif or OpenLook (Section 2.4.12). The layout of widgets in a dialog box is an example of *bin packing*, which is known to be NP-Complete (meaning it is expensive or impossible to find the best solution), so various systems have provided interesting options ranging from simple to elaborate algorithms. DON [Kim and Foley 1993] provided sophisticated 2D layout taking into account issues of balance, effective use of whitespace, and the designer's preferences on sizes, layout, widget choice, and so on. Rather than try to find an optimal layout, it provided the designer with a collection of options from which to choose. Supple [Gajos and Weld 2004] was designed to take into account each user's individual preferences and generate a design at run-time taking those into account, along with the usual criteria such as widget properties and screen size. This would be particularly useful for people with disabilities—for example, by providing large fonts for the visually impaired.

The ITS system from IBM [Wiecha et al. 1990] was notable for its use in creating all the kiosks for the EXPO'1992 world's fair in Spain. The idea was that the vendors, such as the various countries and food places, would specify their own content, and ITS would automatically create the screens from that [Wiecha et al. 1989]. One goal was that all the design rules would be general purpose, and if a particular screen didn't look right, the rules would be modified in a way that would improve that as well as other screens. However, there were many problems in practice: (1) the vendors found the ITS specification language to be too difficult to use, and the IBM researchers had to personally go to Spain and author the content themselves; (2) it was found to be too difficult to author rules that worked for all content, and exceptions had to be made to get specific screens to look as desired; and (3) a rule-based system proved brittle and difficult even for experts to use as rules would interact with each other in unpredictable ways, causing other screens to break when rules were edited to improve a different screen.

Another notable example was the work of my former Ph.D. student Jeff Nichols on the *Personal Universal Controller* (PUC), which investigated moving the control panel for consumer electronics onto one's personal digital assistant (PDA) or smartphone [Nichols et al. 2002, 2006b, 2007]. Among the important advances of this work were generating "personally consistent" user interfaces and combined user interfaces. The first one was inspired by the issue that consumer electronics devices, like copiers, printers, and even alarm clocks, have very different interfaces depending on the vendor and even the model. Most people learn how to use their own copier and clock but may be stymied by another. The PUC showed how to generate an interface for a new appliance that was *consistent* with the UI that people are familiar with, using the same names for functions, menu structures, and screen layouts (see Figure 20.12) [Nichols et al. 2006a]. The second advance was to take the independent specifications of disparate devices, possibly from different manufacturers, and generate a *combined* UI that let the user control them together. The prototypical example of this is a media center, so the user could control a DVD player, the cable box, or the TV using the same UI for all of them [Nichols et al. 2006b]. A summative user study showed that the generated UIs were about twice as fast and successful as the manufacturer's own UIs on their products and four times better when they were *personally consistent* [Nichols et al. 2007]. While comparing to a manufacturer's control panels may be a low bar, this is still the only

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

Figure 20.12 User interfaces generated by PUC for a complex and simple copier without and with consistency: (a) complex copier without consistency, (b) simple copier without consistency, (c) complex copier consistent with simple copier, and (d) simple copier consistent with complex copier. Source: Fig. 1 of Nichols et al. [2006a].

automatically generated UI that has been shown in user tests to be *better* than manually designed UIs.

Whereas model-based design of UIs has not caught on commercially, it has been successful in other areas. For example, software engineers have found it useful when controlling complex processes like the controller of an automobile engine [Hanselmann 2008]. A similar example is that business processes can be specified with a graphical flow diagram, for example, in *Business Process Modeling Notation* (BPMN), and the software automatically generated from that.

20.11 Tools for 3D, AR/VR/XR Interactions

As discussed in Chapter 12, creating three-dimensional (3D) interactions and interfaces is very difficult, so it is not surprising that there have always been many tools that have aimed to help. Some old animation tools, like Macromedia Director, had 3D extensions, but most 3D interfaces and interaction techniques were created in custom tools. For example, Silicon Graphics, an early leader in specialized hardware for graphics (see Section 2.2.2), created GL (for Graphics Language) in the 1980s, to describe 3D scenes, which it modified into OpenGL in the 1990s as a standard. A competitor was the high-end Renderman by Industrial Light and Magic and later Pixar, which has been used for their movies. These languages and toolkits were designed to be used by programmers who had extensive computer graphics training. A current toolkit that is widely used for 3D in JavaScript is three.JS.

Later, there were interactive tools provided for creating 3D interactions. Popular ones today include Unity and Unreal Engine, which have been used extensively to create 3D games but are also used for authoring other augmented/virtual/extended reality (AR/VR/XR) experiences. Often, the characters used in those tools are created with other, more specialized tools such as Alias (1983), which was later provided by Silicon Graphics, then Wavefront, and now AutoCAD. A research system that aimed to make it much easier to create these interactions was the Alice system [Pausch et al. 1995] from Carnegie Mellon University, which evolved into a platform for teaching kids to program [Kelleher et al. 2007]. Today, there are a wide variety of tools to help programmers and non-programmers design and implement 3D interactions, but most still require significant expertise.

20.12 Models and Patterns for the Implementation

When a programmer is implementing a user interface or an individual widget, there are often *software design patterns* [Gamma et al. 1995] or *models* that are helpful. These are general, reusable ways to organize the code. For example, the *command object pattern*, covered in Section 15.12, provides an easier way to implement operations along with the ability to undo and repeat those operations. Some

of these patterns might be classified as *software architectures*—the key difference being the scale—software patterns are often only a small part of the implementation, whereas the software architecture is more about the overall, global design of the whole software system. These models or patterns do not have anything to do with what the UI looks like and are generally only relevant to the software developer (not the UI designer), although some implementation patterns make it difficult or impossible to support some UI designs [Bass and John 2003]. The next sections cover some software models.

20.12.1 Early Models

One of the earliest software models from the HCI community was the Seeheim Model [Pfaff 1985], which resulted from the first UI software tools workshop that took place in Seeheim, Germany, in 1983. A key goal of this model (Figure 20.13) was to separate the code for the user interface part from the code for the application, which generally was thought of as the "real work," or (not particularly helpfully) the code that wasn't for the UI. It was understood that the UI might need to change due to iterative design and user testing, and it was hoped that this could happen without affecting the back-end application code. The user interface code was divided into two parts—the presentation, or what it looked like, and the *dialogue control*, or how it handled the user inputs. (The box at the bottom of Figure 20.13 was never well explained.) Although this model was never used directly, the goals and ideas for how to divide up the software were influential, and many later models also tried to separate out the UI code from the application code.

20.12.2 Event Model

One of the most enduring ways to implement user interfaces is using the *event model*. This means that the software is organized as a collection of units of

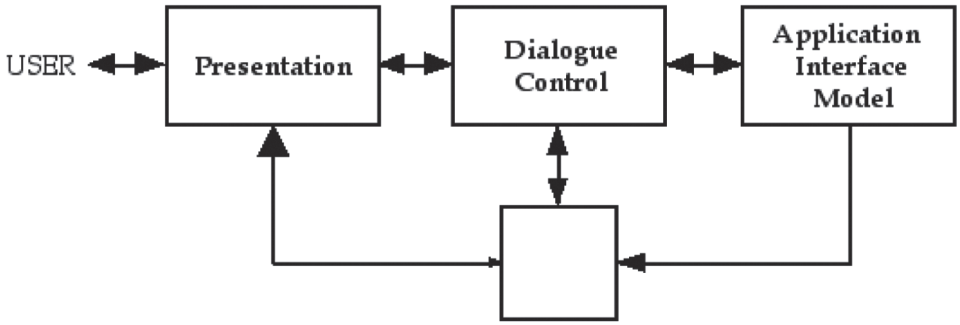


Figure 20.13 Seeheim model. Source: Redrawn from Pfaff [1985, p. 9].

code (called the *event handlers*) that are called by the system when a user or system triggers an action (the *event*). Examples of typical events include mouse-left-button-down, mouse moved by an incremental amount, keyboard-key hit, and also system-generated events like a window becoming uncovered. Many toolkits also use the event model to handle when the user operates menus and other widgets, so there will be events for *menu-item-picked*, *scroll-down*, or *window-close-icon-picked*. Consequently, there may be a large number of different types of events that the programmer may need to handle. The X/11 system (Section 2.4.12) from 1987 defined 33 types, Windows .Net from about 2000 defined 69 types, and JavaScript now describes 52 document object model (DOM) events, but this is a misleadingly small number because each of the JavaScript events has subtypes—for example, the DOM `MouseEvent` has *10 subtypes*.

All of these events will have *parameters* used by the code, such as the X and Y locations of the mouse click, which key (like "m") and whether any modifier keys (like Shift, Ctrl, Alt, etc.) were down when a keyboard key was hit, or which menu item was selected. In older systems, like X/11, Windows, and Macintosh, the events were implemented with fixed-size data structures (e.g., overlaid variant records in C) with a size to fit the event with the largest number of parameters. In current systems with dynamic memory, like Java or JavaScript, the event structures can be dynamically sized by the actual parameters used for each type.

Custom languages with built-in event handlers have been designed for both research and commercial systems. Examples include Sassafras [Hill 1986], HyperTalk (Section 2.4.9), Microsoft's Visual Basic, the Lingo scripting language in Macromedia (later Adobe) Director, Adobe Flash, and HTML/JavaScript. Other languages, such as C++ and Java, figured out ways to provide an event model with conventional languages. Some of the custom languages, notably Sassafras and HyperTalk, embraced the event model for the *entire* design, so the key control structure used by the programmer was to raise new events inside the event handlers for other events. For example, the event handler for a menu item might raise a *change-line-style* event, which the programmer would then write a handler for. In most modern systems, in contrast, the programmers use only the built-in, toolkit-supplied events, and use regular methods for other control flow.

A key distinction for event handlers is what they are attached to. Some early event models just had global handlers or had all handlers attached to the window, but all modern systems allow event handlers to be attached to lower-level objects. For example, a button or even a rectangle might have a click event handler attached to it, which is only invoked when the user performs a click while the mouse cursor is pointing to that particular object. If the object at the cursor location does not have an appropriate handler (or the handler passes the event through), then the event

propagates to other objects to see if they have appropriate handlers. Typically, this propagation goes up the container hierarchy (to the group this object is in) recursively until the window (or document) object is reached. (However, JavaScript has particularly complex propagation, which goes *both* down and then up the hierarchy looking for event handlers.) As discussed in Sections 8.7 and 11.4, keyboard events generally follow the *input focus*, which will be attached to different text input fields based on user clicks or using the "Tab" keyboard key, rather than following the mouse cursor.

Even though the event model has stood the test of time and is still the leading way that UIs are implemented, it does have a number of well-known issues. The key one is that it can be difficult for the programmer to understand the flow of control and to debug when incorrect event handlers are invoked. As mentioned above, it is complex to determine which handler will be called for an event, and then, after the event handler happens, it is especially difficult to determine what happens next—for example, does that event handler trigger a different handler to happen? There are often global state variables that control what an event does, such as indicating whether any objects are selected, which also complicates the flow of control. A related complication is that event handlers generally communicate to each other through global variables, which can also be difficult to track and debug. For example, a move event might be dragging around a graphical object, but which object is being dragged is typically stored in a global variable that would be set by the handler for a mouse-down event. Another difficulty is even understanding which events are generated. For example, when the user double-taps on a mouse button, that generally will cause the following events, in this order: mouse-down, mouse-up, mouse click, mouse-down, mouse-up, mouse-click, double-click—so the programmer must be prepared for the event handlers for all of those events to be executed on double-click. It is even more complex when a user taps with a finger on a phone since (in most current toolkits, including at least JavaScript) tap events are different from mouse events, but tapping the screen with a finger triggers *both* tap events *and* mouse events for backward compatibility.

20.12.3 Listener Pattern

One way to provide the event model is using the *listener pattern*, which is where a number of objects (the *listeners*) are attached to objects where the events happen, and appropriate methods of the listener objects are called automatically when the events happen. This is used most classically in conventional languages where events are not built in. For example, in Java, one defines classes that implement the `MouseListener` interface and then attaches instances of those classes to windows (called `Frames` in Java) or other graphical objects:

```
public class MouseListenerExample extends Frame implements MouseListener{
    MouseListenerExample(){
        addMouseListener(this);
        ...
    }
    public void mouseClicked(MouseEvent e) {
       ...
    }
    ...
public static void main(String[] args) {
    new MouseListenerExample();
}  }
```

A key advantage of the listener pattern is that the object on which the events happen is decoupled from the objects that want to act on the events, and the code can be compiled separately even though the object that receives the events calls the methods of the listener when the events happen. Another advantage is that there can be *multiple* handlers attached to the same object, of the same or different types.

The listener pattern in an example of the *observer pattern* (one of the twenty-three well-known *Gang of Four* design patterns [Gamma et al. 1995])—but the observer pattern is more general since it allows *any change* of the observed object to trigger the calling of a method of the attached listeners, and not just specific events. For example, the observer pattern can be used to have a controller be notified when the model changes its value in the MVC model described below (Section 20.12.6).

20.12.4 Interactors Model

Inspired by the concept that some standard behaviors are repeated in many different contexts, as proposed in the early Foley and Wallace 1974 *virtual devices* [Foley and Wallace 1974] and 1984 *interaction tasks* [Foley et al. 1984] (see Section 19.5.1), I developed the *interactors* model in 1989 as a way to try to make it easier for programmers to implement new widgets [Myers 1989, Myers 1990a]. The key principle is that there are a small set of predefined, highly parameterized *interactor* object types, which the programmer can create instances of to achieve most of the standard behaviors. These are attached to graphical objects. The idea is to completely eliminate the need for writing any event handlers. This also serves to separate the implementation of the behavior (response to inputs) from the graphics (or, in terms of the MVC pattern described below (Section 20.12.6), interactors implement the controller independent of the view). Interactors were first implemented in the Garnet toolkit in Lisp (1987–1994) [Myers et al. 1990] and then re-implemented in the

Amulet toolkit in C++ (1994 and 1997) [Myers et al. 1997a] with extensions to support command objects and selective undo (Section 15.12) [Myers and Kosbie 1996]. The basic kinds of interactors include:

- *Menu (Choice) Interactor*. For choosing one or more items from a set, or for a single standalone button.
- *Move-Grow Interactor*. For moving or changing the size of an object or one of a set of objects using the mouse. This interactor can also be used for 1D or 2D scrollbars and horizontal and vertical gauges, and for moving or growing application objects in a graphics editor.
- *New Points Interactor*. For entering one, two, or an arbitrary number of new points using the mouse—for example, for creating new lines, rectangles, or polygons in an editor.
- *Angle Interactor*. For calculating the angle at which the mouse moves around some point. It can be used for circular gauges or for rotating objects.
- *Gesture Interactor*. For capturing all the points the mouse goes through between start and end events, and recognizing them as a gesture (Chapter 14). This interactor can also be used to just capture the points as needed for freehand drawing (also called *inking*).
- *Text-Edit Interactor*. For inputting a small (optionally multi-line) string of text.

The parameters available for controlling the interactors include what graphics they are attached to, which events (e.g., mouse buttons) start, stop, or abort them, and interactor-specific parameters. For example, the Menu (Choice) Interactor has a parameter for whether the user can select zero, one, or multiple items (so it can be used to implement regular buttons, radio buttons, or checkboxes, Section 7.3.1), and the Move-Grow Interactor has a parameter to control gridding (Section 10.6.5).

Although the Interactor model is useful for implementing widgets, as discussed in Section 20.13, that is not often needed, so there has been little uptake of this model. One place where the interactor model was used was in Adobe Flash Catalyst, from 2011–2012 (Figure 20.9), which was specifically designed to help build new widgets in Flash.

20.12.5 Imaging Model

As defined in Sections 11.2 and 20.3, the *imaging model* is *how the graphics and text are drawn on the screen*. In the context of this chapter, the imaging model is how the programmer is required to draw the graphics. Typically, there are a large number

of different methods available, like DrawRectangle, DrawEllipse, DrawLine, DrawText, DrawImage, and so on. Each of these will typically have a large number of parameters, like the location and size to draw the shape, the color, line thickness, and so on. There will also be global parameters, such as a bounding-box in which to clip the graphics, settings for the overall coordinate system and transformations (such as scaling), and so on. The actual algorithms for how these work (that is, which pixels to turn on to make a shape appear) are covered in graphics textbooks [Newman and Sproull 1973, Foley et al. 1990, Hughes et al. 2013].

When the first bitmap displays were being designed in the 1970s (see Section 2.3), memory was very expensive, so the Alto (Section 2.4.3), Star (Section 2.4.4), Macintosh (Section 2.4.7), and early workstations like the Sun and PERQ (Section 2.4.12) were all monochrome—each pixel was represented by 1 bit and was either black (0) or white (1). For example, the Alto screen was 606 by 808 pixels, so that required 61,206 bytes of storage, which was almost equal to the 96K of storage the original machine had for all of its software!

The imaging model for monochrome displays focused on drawing simple primitives and copying rectangles of the screen around using an operator called *BitBlt* or *RasterOp*, which could also combine the source rectangle into the destination rectangle using any Boolean operator (like OR, AND, NAND, or XOR). Early researchers identified clever ways to use BitBlt to achieve other operations. For example, BitBlting a monochrome rectangle using XOR with a rectangle of all 1's will reverse all of the pixels, very quickly toggling white and black without the need to redraw any text or images in the rectangle. This was used, for example, to highlight the selected items in menus in Smalltalk (Figure 2.10(b)) and anything selected, including menu items and icons, on the Lisa and Macintosh (Figures 5.11 and 5.12). The item could be deselected by XORing with the same mask, again without redrawing anything. The Lisa and Macintosh grayed out illegal items, like in menus (Figure 5.11), using halftoning—meaning that a mask rectangle with a pattern of pixels off and on was BitBlt with the target area using the AND operation. This did not require redrawing to gray-out the item, but it did require redrawing to make it enabled again (but objects did not typically change from disabled to enabled much, unlike changing the selection).

Systems that relied on these "tricks" were not easy to move to color displays with multiple bits per pixel, where XOR and AND produce unpredictable other colors (not the "reverse" colors). A story is that one reason that HyperCard (Section 2.4.9) did not catch on is that its imaging model, including animated transitions and paintings, was optimized for monochrome graphics and did not transition well to color screens. Similarly, the Motif window manager was designed to look good in color, whereas OpenLook was designed based on the monochrome Star look

and feel (Section 2.4.12)—one reason Motif was preferred as everyone transitioned to color screens. Another story is that when Steve Jobs was designing the NeXT (Section 2.4.11) starting in 1985, he anticipated that future machines would be in color, but he couldn't afford color hardware, so he used a gray-scale monitor, with 2 bits per pixel, to make sure that developers did not rely on the tricks that only work with a monochrome imaging model.

Today's imaging models provide elaborate color models for defining the graphics being drawn, including color gradients and textures, and of course, special graphics hardware on today's computers enables real-time 2D and 3D graphics and even video to be drawn quickly.

Besides colors, programmers have a wide variety of low-level details that they need to deal with in imaging models, which are surprisingly complex and intricate. For example, consider which pixels will be drawn for the following call:

```
DrawXXX(3,4,10,15)
```

It is clear that it starts at pixel (3,4), but if it is a draw*Line* call, then the line goes to pixel (10,15), but if it is a draw*Rectangle* call, then it has a *width* of 10 pixels, and goes to pixel *12*. (You might expect that the last pixel would be $3 + 10 = 13$, but that would be an off-by-one error since then there would be 11 pixels on instead of 10!) If you add in the complexity of drawing thick lines, it becomes even more complex because most imaging models draw on *both sides* of the main coordinates (that is, both up and down for a thick horizontal line), whereas other imaging models draw only inside (i.e., does a rectangle grow both out and in or only in as the border's line width gets thicker). There are also parameters that control what the dash pattern looks like for dashed lines, how the end caps are drawn for thick lines, how the intersections of polylines look, whether filled self-intersecting polygons like the 5-pointed star fill the center or not, how curved lines are defined, and many others. For drawing text, there is another whole set of parameters that a programmer can set and/or read, including the font, style, and many aspects of the character or whole text sizes (ascent, descent, leading, kerning, em-width, baseline, etc.). Anti-aliasing is a graphical technique to make graphics look less jagged by blurring the edges of characters and shapes, and it adds even more complexity to some of the implementation decisions.

Another interesting difference among imaging models is how the programmer is required to specify all these parameters. There are too many to just pass them all into the `Draw-XXX(…)` operation (e.g., there are more than 20 parameters that control how a line is drawn), especially since most of the parameters are numbers so the programmer would be likely to mix up which are which. There are three main models that have been used instead:

- An object, often called the *style* or *graphics*, that stores many of the parameters. This model is used by Java Swing (the *graphics2D* object) and HTML/JavaScript SVG (for Scalable Vector Graphics) (the *style* of the graphics). There can typically be more than one of these objects, and it is provided as a parameter to the graphics operations along with the "basic" parameters like the start and end of a line, like:

```
drawLine(3,4,10,15,"stroke:blue;stroke-width:2")
```

- The *pen* model, where there is a global state, called the *pen*, that remembers the state, and the programmer typically precedes drawing with setting many different aspects of the pen style, usually each with its own separate call. This style is used by the Macintosh and JavaScript Canvas. The code for this looks like:

```
setColor(blue)
setStrokeWidth(2)
moveTo(3,4)
drawTo(10,15)
```

- A programming language model, which has only been used by PostScript and its derivatives such as NeWS (Section 2.4.12) and Display PostScript used by NeXT (Section 2.4.11), where the programmer generates the code to cause the correct graphics to be drawn.

20.12.5.1 Retained Object Model

A key aspect of the imaging model is whether it uses a *retained object model* (also called a *display list*) versus a painting model. In the retained object model, the system keeps a list of what will be displayed on the screen, and the programmer changes the display by editing that list. In contrast, in the painting model, the programmer controls what is on the screen by directly painting it using various kinds of drawing[4] calls.

4. As mentioned in Section 20.12.5, the procedures to cause graphics to appear on the screen are often called *draw-xxxx* in many toolkits. In the terminology of end-user graphics programs described in Section 2.5.1.5, which differentiates *painting* end-user applications like MacPaint, Microsoft Paint, or Adobe Photoshop from *drawing* applications like MacDraw, Microsoft PowerPoint, or Adobe Illustrator, it would be more correct to say that these procedures cause *painting* to happen since drawing programs keep a display list of the objects on the screen that the user can edit, but procedures like `drawRectangle` do not keep track of what is drawn.

Both models have been used by many systems dating back to the beginning of graphics. The vector terminal used by SketchPad (Section 2.4.1) in 1963 used the retained object model, as do all character terminals (Figure 2.7). The early bitmap screens on the Alto, Star, Macintosh, and Windows machines all used the painting model. Today, the Macintosh, Windows, iOS, and Android all use the painting model at a low level and the retained object model for the toolkits of widgets (programmers who need to control which pixels are displayed, for example, to implement widgets, call draw-xxx methods, whereas programmers who use collections of widgets just create the widgets, specify widgets' parameters, and change the screens by editing the list of widgets).

On the Web, the DOM itself is a retained object model, which programs can edit. Within the DOM, for graphics, a programmer can choose to use a Canvas element, which supports the painting model, or SVG, which provides an area of the screen with a retained object model for graphics. Both retained object and painting models are available for programmers doing 3D drawing as well (Chapter 12). Research systems have also explored both models. My Garnet [Myers et al. 1990] and Amulet [Myers et al. 1997a] toolkits both focused exclusively on the retained object model since it makes it easier for constraints and interactors to have objects to connect to.

20.12.6 Model–View–Controller

The MVC paradigm [Krasner and Pope 1988] was one of the very first software design patterns described as a way to make it easier to implement user interfaces, beyond just using OO programming with inheritance. It was developed in the early 1980s in Smalltalk (Section 2.4.3.1) with the goal of better modularizing the UI software. A key goal [Krasner and Pope 1988], which continues to motivate the design of most software architectures and models, is to better separate out the code and data for the application (here, the *Model*[5]) from the code and data for the user interface. A further goal of MVC is to separate out the graphics (*View*) from the behaviors (*Controller*). Typically, instantiations of the MVC pattern in different toolkits have additional benefits such as supporting *multiple* views of the same model and providing complete separation of the code for the model from the code for the UI using the observer pattern (Section 20.12.3).

The classic flow of control with MVC is that if a program updates a value, say what part of a document a window is showing, the code would update the model—for example, that lines 200 through 280 are displayed. This would automatically trigger some listener code to update the views, which would cause the graphical

5. Note that this use of the word "Model" is different from other uses in this and the previous chapters, as discussed in Section 19.2.

display to change—for example, to change which lines are shown in the window and where the indicator is in the scrollbar. If the user interacts with the graphics, for example, clicking on the scrollbar, this would be handled by the controller code, which would identify the corresponding changes to the model (e.g., that now lines 210 through 290 are displayed). When the controller updates the model, this would automatically cause the views to change, as mentioned above.

Smalltalk had the earliest and most famous implementation of MVC, but it has been supported by almost every toolkit since then. For example, the Apple toolkits have whole collections of classes that inherit from `NSView` and `NSController`, which are designed to be connected to the programmer's models.

A story about modern toolkits is that the original AngularJS toolkit from Google (from about 2009) relied on the MVC paradigm, but when React was being developed by Facebook around 2011, it explicitly rejected MVC due to MVC's requirement of copying values around (typically from the model to the view and from the controller to the model) using constraints or the observer pattern. Instead, React relies on shared data, which it calls *states*, that are accessed by all the code that needs them. This is credited with part of the significant speed advantage for React (along with the absence of regular constraints (Section 20.6.2), which motivated the redesign of AngularJS into what is called now called just "Angular" around 2016). However, other web frameworks, such as VUE, do support MVC.

20.12.6.1 Model–View

Although it is often clear how to separate the code for a model from the many ways it can be displayed and manipulated, it turns out to be almost impossible to separate the code that implements a behavior (controller) from the graphics that the behavior affects (view). For example, a classic scrollbar (see Figure 5.12(b)) has specific and different code for handling mouse events on the arrow icons at the end, in the trough, and on the thumb. And this code is entirely different from the code for handling scrolling using the "Page Down" button (Section 5.5) or autoscrolling due to selection or searching (Section 5.2), all of which are supported by text editors. Consequently, in most implementations of MVC, one will find a different xxx-View for each xxx-Controller class, where the code in those two classes is tightly coupled.

Therefore, a number of systems just use a Model and a combined View–Controller, including the Andrew Toolkit [Palay et al. 1988], where the combined View–Controller is just called the View, and in the internal design of Java Swing, where the View–Controller is sometimes called the "Delegate" or just the "View" [Fowler 1998]. This revised paradigm is sometimes called the "Model–View" pattern.

20.12.7 Document Model

Another software pattern to help programmers implement interactive applications is the *document model*, which was first provided by Smalltalk but more importantly promoted by MacApp from around 1986 [Schmucker 1986], and still provided by (at least) Apple's UI frameworks (Section 20.5.2).

From the end-user's point of view, the document model supports *applications that operate on individual documents, which are represented as individual files*. Examples include text editing programs (Microsoft Word, Apple Pages), graphics programs (Apple MacDraw and MacPaint, Microsoft PowerPoint, Adobe Illustrator, and Photoshop), spreadsheets, and code editing tools. Applications that do *not* match the document model include browsers, games, and many Internet-based applications like a reservation booking app.

From the programmer's point of view, the *document model* is part of a framework, and it provides high-level classes, like `Document`, that the programmer subclasses to create document model applications. The `Document` class provides implementations for the generic file operations, like opening, closing, managing multiple windows, and so on, and provides empty placeholder implementation for standard printing, editing, formatting, and viewing the contents of a document. Section 15.12 above mentioned the command object pattern, which is often supplied with the document model to support undoable operations. The programmer creates subclasses that override these high-level placeholder methods to support the actual operations.

Note that this model is different from Web DOM, which uses the word *document* in a different way.

20.13 Considerations for the Implementation

This chapter has covered a wide range of implementation considerations at many levels for many kinds of tasks. In summary:

- For prototyping either using or creating widgets, consider using interactive prototyping tools, which may have many built-in prototypes of widgets you can start from (Section 20.9).
- For the programmer *using* interaction techniques:
    - Different platforms (Mac, Windows, Android, etc.) and tools (Angular, React, etc.) have different collections of widgets, so that is usually the first consideration.
    - Use high-level frameworks (Section 20.5.2), if available, since these will decrease the amount of code needed, especially if you can leverage

models like Documents (Section 20.12.7) and Command objects for undo (Section 15.12).

- A first design step is to choose which widget is most appropriate for your context, such as how much space is available, how many items to choose from, or how much text needs to be entered. See the guidance for this in the chapters for each kind of widget.
- If the layout of the screens will be mostly static, then using interactive GUI layout tools may be sufficient (Section 20.8).
  - Alternatively, if the layout or contents of the widgets need to depend on values or text from other fields, or based on dynamic run-time data, then a layout manager (Section 20.6) may be required.
    - ✓ There may be interactive tools that help with using a layout manager as well, depending on the platform.

- For the programmer *creating new* interaction techniques and tools for them (creating new interactive tools, toolkits, or other APIs):
  - It is important to verify that a new widget is *really* needed since the supplied widgets are usually carefully designed, and creating your own may result in inconsistencies with the way other applications work.
  - A related issue is that widget implementation is very difficult, with intricate and often extensive code required. Many different states and transitions are often needed (see Section 20.2), so it might be helpful for the implementer to consider creating a transition model (Section 19.4). For example, many Flash and HTML implementations of menus omitted handling of pressing inside and releasing outside.
  - The implementer should also ensure that the widget behavior can be aborted and undone; that the widget can be operated with the keyboard (e.g., with arrow keys), with a mouse, and with a touch screen; and that it is accessible.
  - We have noticed that very few people actually know the various ways that widgets can be used and the available interaction techniques (e.g., many people don't know that iOS interactions can sometimes be undone by shaking the device). Therefore, the designer must consider how to make the interactions *discoverable*.

- For all programmers of UIs, various patterns and models (Section 20.12) have been provided that may make the programming easier but also may make it more complex—these are appropriate to use when their benefits are important. For example, the MVC pattern provides better separation between parts of the code but causes extra run-time performance overhead and requires more code, which can be more difficult to understand and debug.

20.14 Chapter Conclusions

Implementing user interfaces has always been difficult. Having a library of widgets to use in code is a big help, but still much complexity remains. Creating *new* interaction techniques requires significant attention to detail to ensure that they work in all situations and are consistent with users' expectations.

Research has long focused on tools to help with implementations, both interactive GUI tools and programming libraries and frameworks, but it is difficult to study and quantify how much such tools help [Myers and Stylos 2016]. It is clear that without the early research on tools (see Section 2.5.2), it wouldn't be as easy as it is today to implement UIs [Myers 1998a, Myers et al. 2000a]. Today's tool research is helping to determine how future UIs—such as those using 3D, VR, and AI—will be built since today's tools cannot be used directly. However, such research naturally lags research into the new styles. Usually, it takes until a style has become routine (like GUIs are today) for the tool makers to figure out the best way to provide libraries to support them.

CHAPTER 21

Methods for Designing and Evaluating Interaction Techniques

21.1 Introduction

If one is studying which interaction techniques to use or else trying to create a new interaction technique that is "better," there are a wide variety of human–computer interaction (HCI) methods one might use. As I have discussed in some of my papers [Myers et al. 2016], there are many methods that are applicable to the entire process—from needs-finding and requirements elicitation, to design, implementation, and informal and formal evaluations. Whole books are available that discuss these methods (e.g., Nielsen [1993], Hartson and Pyla [2012], Lazar et al. [2017], Shneiderman et al. [2017]), including books that focus on a single method (e.g., Beyer and Holtzblatt [1998]), so I can't do justice to them all or even describe one in sufficient detail that this chapter can serve as a sufficient resource. However, it still seems useful to summarize what methods I have seen used in the design and evaluation of interaction techniques.

21.2 Methods for Needs Finding

There are many methods that help the designer better understand the needs and requirements of the users or other stakeholders. For example, in my research in various domains [Myers et al. 2016], we frequently use *contextual inquiry* [Beyer and Holtzblatt 1998] to understand what users are doing as part of their activities, along with interviews and surveys to understand people's self-reports about their needs and opinions, and repository studies to understand current code and outputs.

However, specifically for interaction techniques, there does not seem to be as much need for these formative, early-stage methods. The problems that a new interaction technique are trying to solve are generally clear.

21.3 Methods for Design

When *designing* a new interaction technique, the designer is likely to create a variety of options to explore various looks and feels. Whereas exploring various *looks* for an interaction technique is fairly easy (see next section), exploring the "feel" or *behavior* usually requires writing code since few interactive tools support more than changing the look on clicks (*click-through prototypes*) or simple animations, and tools like Adobe Flash Catalyst that tried to allow designers to directly explore behaviors (Section 20.8) ended up failing. However, as discussed in Section 1.8.2, designing the behavior of interaction techniques is especially difficult and important.

21.3.1 Graphic Design

Having a high-quality *graphic design* is important for interaction techniques. Why the *look* is difficult to design was covered in Section 1.8.4. Graphic design specifically for menus was discussed in Section 7.3.7. A key challenge for the graphic design, besides aesthetics, is to make it obvious what the user can do with the widget by providing good affordances (see also Section 1.3.2). Making widgets look more three-dimensional (called *skeuomorphism*) was supposed to help with making their affordances more obvious. This look was probably first explored in the Alternate Reality Kit from 1985 (see Section 2.4.10) [Smith 1986, 1987] and in the NeXT (Section 2.4.11), and then became widespread, but fell out of favor around 2012 when a more "flat" design became predominant.

A graphic designer would typically draw low-fidelity or high-fidelity versions of the look of the interaction technique, often using interactive prototyping tools (Section 20.9) like Adobe Photoshop, Figma, or Balsamiq.

It has been argued that a focus on the "beautiful designs" and aesthetics of the iPod and iPhone by Jonathan Ive, Apple's senior vice president of industrial design (and later Apple's Chief Design Officer), was a key factor in their success [Grudin 2017, p. 105]. However, as I have argued in a few chapters, I believe that today's visual design has shifted too far toward being pretty such that it is lowering the usability. For example, the current scrolling techniques do not provide easy ways to get to the top or bottom of the document or to move by pages (Figure 5.16(a)), and today's flat graphic style reduces the affordances for knowing what is interactive [Norman and Tognazzini 2015].

21.3.2 Prototyping and Implementing

As with any other kind of user interface, it is critical for interaction techniques to be iteratively prototyped and evaluated with target users. The methods and tools for doing this are also not much different than for other UIs. Tools for prototyping and

implementing were covered in Chapter 20, and tools and methods for evaluating the prototypes and implementations are covered next.

Unfortunately, it is often not possible to adequately test a low-fidelity or even a higher-fidelity prototype of interaction techniques since their effectiveness often depends on factors that may only be apparent in the real implementation, such as responsiveness and accuracy. For example, the flick scrolling interaction (Section 5.9.1) works effectively on an iPhone due to the specific speeds and decelerations in the Apple implementation, whereas I experienced an early Nokia smartphone implementation of flick scrolling that was unusable due to feeling too sluggish (it was like trying to push the content through mud). Another example is that evaluations for a gesture recognition interface (Chapter 14) will depend entirely on how accurate the recognition is, which will only be apparent from the real implementation.

21.4 Methods for Evaluation

After a prototype or implementation of an interaction technique has been developed, it is crucial that it be evaluated. The methods for evaluating any UI can be applied to interaction techniques, such as a *usability evaluation* [Nielsen 1993] to test whether users can use it as expected, and *controlled experiments* (sometimes called A/B testing or, in the most formal case, *randomized controlled trials*, RCTs) to more formally test whether one design is better than some other design (which may be the previous version, or two alternative new versions).

21.4.1 Usability Evaluation

Usability testing is a less formal way to test to what extent people can understand how to use the interaction technique as expected. It includes such aspects as (see also Section 1.3.2):

- *Learnability*—how well people can learn how to use it, with the expected support (documentation, help systems, etc.).
- *Discoverability*—how well people can notice the interaction techniques, and figure out how to use it on their own.
- *Error-proneness*—how often do users make mistakes when using the interaction technique, and to what extent do they notice when the mistakes are made, and figure out how to recover from the mistakes. This is one aspect of the *reliability* and *accuracy* of the interaction technique.
- *Effectiveness*—how well the interaction technique can be used by most users. The user's *performance* and *speed* are part of this.
- *Satisfaction*—how much do people *like* the interaction technique, or conversely, what is annoying about it.

There are a wide variety of methods for measuring usability. *Discount* measures [Nielsen 1990] involve experts evaluating the interface using a set of guidelines, like Nielsen's famous 10 "heuristics" (see Figure 21.1) [Nielsen 1994] for *heuristic evaluation* (HE), or going through specific tasks to evaluate how difficult the expert thinks each step will be in a *cognitive walkthrough* [Lewis et al. 1990]. Alternatively, the interaction technique might be evaluated based on platform or web guidelines, like Apple's Human Interface Guidelines [Apple Computer 1992]. These assessments are most effective when performed by experienced UI specialists.

However, the most valuable usability method involves putting the interaction technique in front of actual people and seeing what happens. The above aspects can often be observed by the experimenter watching the user trying to do realistic tasks with the interaction technique, and in some cases, they can be counted or measured to get numbers (e.g., how many errors per session did users make). A key goal is to understand the problems and successful parts of the interface so that they can guide future improvements. Often, users will be asked to *think aloud* so that the experimenter can gain insights into what the user is trying to do or is confused about, and *why* the user is doing things. These user studies can take place in a lab, remotely through screen-sharing, or even asynchronously with software that logs what the user does. Often, a questionnaire or survey is involved after the hands-on portion so as to assess the user's satisfaction, which might use a standardized questionnaire like the *System Usability Scale* (SUS) [Brooke 1996] or the *Questionnaire for User Interface Satisfaction* (QUIS) [Chin et al. 1988].

Another dimension for physical devices, such as pointing devices, but also physical buttons, knobs, sliders, keyboards, and so on, is how *comfortable* they are to use. This is sometimes called the *ergonomics* or *human factors* of the devices. As mentioned in Section 17.3.1, one goal is to avoid repetitive stress injury and other harm caused by long-term use.

Finally, all interaction techniques should be assessed for *accessibility*—will people of all abilities be able to use them successfully? Often, this assessment uses expert analyses against a set of guidelines, like the Web Content Accessibility Guidelines, but it is also a good idea to test the interaction technique with users who have a wide variety of abilities (see also Chapter 17).

21.4.2 Controlled Experiments

Since most academic papers presenting new interaction techniques want to provide evidence that the new one is better than existing techniques, *controlled experiments* are the norm. These produce *numbers* to tell how *much* faster or more accurate the new technique is compared to the control, and these use statistics to measure how likely the differences are due to random user variations or are really

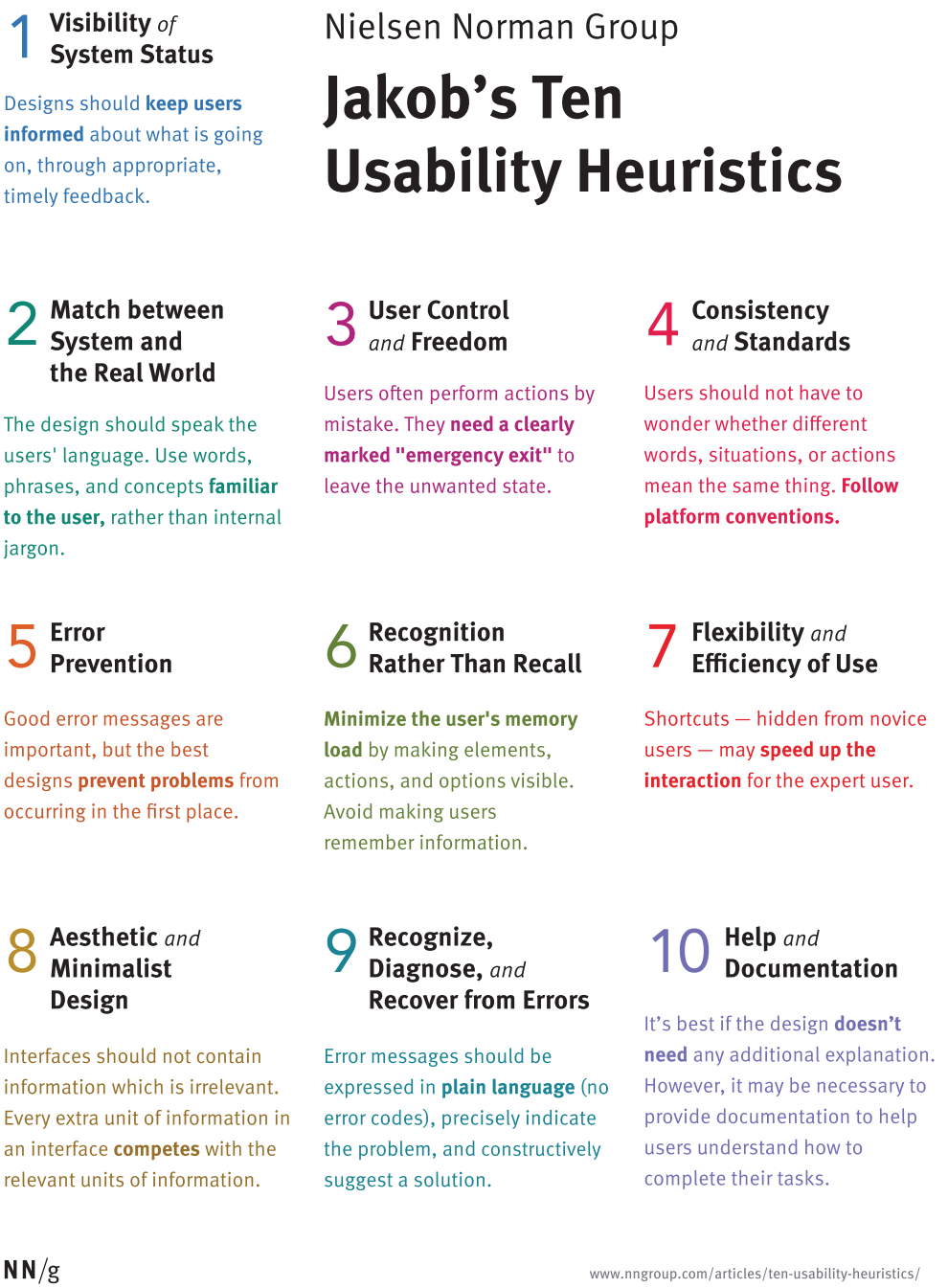


Figure 21.1 Nielsen's 10 heuristics. Source: Image courtesy of the Nielsen Norman Group and Jakob Nielsen, https://media.nngroup.com/media/articles/attachments/Heuristic_Summary1-compressed.pdf.

due to the new technique being better. Some use of these evaluations in commercial products has also been reported, dating back to the extensive studies about many aspects of the design of the Xerox Star (Section 2.4.4) [Bewley et al. 1983].

For all of these kinds of evaluations, a question is what the user should be asked to do, and what to measure, which usually includes *speed* and *accuracy*, and sometimes includes some of the aspects in the previous section, like *learnability*. This book has also presented specific evaluation metrics for specific kinds of interaction techniques that help determine what to measure and how:

- *Pointing devices* (Chapter 4) are often tested for speed and accuracy using a Fitts' Law test (Section 4.14).
- *Scrolling* (Chapter 5) is actually rarely measured, and there has been no standard way to evaluate the performance of a new way to scroll, although we have proposed one in our recent paper [Chen et al. 2023]. Some of my old papers have measured the combination of scrolling and pointing by seeing how quickly users could scroll and then click on targets at various locations throughout a long document [Buxton and Myers 1986, Myers et al. 2000b].
- *Menus* (Chapter 7) have a variety of relevant properties that can be measured, including the findability of a desired item and the accuracy and speed with which users can select a desired item. Examples of the use of these methods were presented in Section 7.5 about methods used to evaluate how the contents of menus should be arranged.
- *Text entry* (Chapter 8) has extensive research on the appropriate methods for evaluating a new technique, as covered in Section 8.5, which discusses a variety of methods for measuring text entry speed, learning, and accuracy.

For the other interaction techniques, measuring how fast and accurately users can perform target tasks are generally used.

21.5 Considerations and Chapter Conclusions

Since interaction techniques are designed to be used across multiple applications and in multiple contexts (at least according to my definition), the usability requirements on them may be higher than for other kinds of user interfaces. Therefore, using appropriate methods for their design and evaluation can be important and impactful. It was not intended that this chapter teaches all one needs to know about this topic but rather to provide an overview and pointers so the designer can know where to start. Of course, ideally, trained and experienced usability experts would be involved since research has repeatedly shown that they do a better job, even compared to others using the correct methods.

CHAPTER 22

Legal Issues and Patents

22.1 Introduction

This chapter is about *intellectual property* and how it can be legally protected in the United States and elsewhere. But first, the required disclaimer: although I have personally participated in a number of legal matters, both as an expert witness (which means I was a consultant to lawyers to help them understand and explain the HCI aspects) and as a co-inventor myself on three issued patents (see Section 1.5), I am not a lawyer, and this chapter should not be considered legal advice.

But what is intellectual property? First, *property* is something that can be owned. Then, intellectual property is property that is intangible and results from creativity. The chief kinds of intellectual property discussed in this chapter are *Copyrights*, *Trademarks*, *Trade Secrets*, and *Patents*, each of which has a subsection below.

Why have a chapter here about intellectual property? As mentioned in Section 1.6, patents on interaction techniques can have enormous monetary value, such as Apple's patent on the way that the iPhone scrolls (discussed in Section 5.9.1), so the designer of a *novel* interaction technique might want to consider getting a patent on it. Other patents are discussed throughout this book (e.g., Sections 3.13, 9.4, 14.4, and 16.3). Trademarks have also had impact on the naming of interaction techniques and devices (e.g., the Palm Pilot and Palm PC discussed in Sections 3.13 and 3.14).

Further, developers *using* interaction techniques may need to consider whether they are violating any intellectual property. Generally, if one is using the standard widgets (like in HTML) or a toolkit from a company, like Microsoft's or Apple's, that company has given you permission to use any of their intellectual property embodied in those widgets. Also, as presented in this book, many interaction technique designs are older than the lifespan of patents, which is 20 years, in which case the patents have expired, and one is free to use whatever is described.

Note that this chapter focuses on the legal aspects of intellectual property in the United States. Other countries and regions (like the European Union) have similar, but often not identical, laws about these topics. The ability to own intellectual property in the US is enshrined in the US Constitution: where Article 1, Section 8, Clause 8 says:

> [The Congress shall have Power ...] To promote the Progress of Science and useful Arts, by securing for limited Times to Authors and Inventors the exclusive Right to their respective Writings and Discoveries. https://constitution.congress.gov/browse/article-1/section-8/clause-8/

22.2 Copyrights

A *copyright* gives the owner exclusive rights to the exact expression of an original work for a period of time, currently about 100 years in the US after the author dies. Copyright does not protect the *ideas* or information behind something, only the specific *expression* of those ideas. For example, the idea of dragging a file to an icon cannot be copyrighted (but it might be able to be patented); however, the particular *expression* of that idea is copyrightable. For example, in the famous legal case *Apple Computer, Inc. v. Microsoft Corporation and Hewlett-Packard Co.* (on which I served as an expert witness consultant for Apple), the court ruled that Apple could not copyright the general idea of a GUI or the desktop metaphor. However, Apple was the first to use a rendering of a trashcan (see Figure 22.1(a))—since the Xerox Star, for example, had used the keyboard "Delete" key for that operation (see Figures 2.14 and 9.1 vs. Figures 2.15 and 5.12(a)). Therefore, Apple could copyright the trashcan, which is why Microsoft uses a "recycle bin" (Figure 22.1(b)). See also Norm Cox's perspective on this case in Section 26.3.

Books, music, poems, articles, lecture slides, and other writings are obviously copyrightable (like this book!), as are paintings, drawings, icons, photographs,

(a) (b)

Figure 22.1 Icons representing how to delete files, from (a) the original Apple Macintosh and (b) Microsoft Windows. Sources: Screenshots, Microsoft product screenshots reproduced with permission from Microsoft.

and other pictures. Computer program code is copyrightable, but again, not the algorithms or concepts that the code implements, only that particular expression.

A controversial issue around computers and copyrights is whether an application programming interface (API) can be copyrighted. In the legal case *Oracle America, Inc. v. Google, Inc.*, at issue is the definition of the Java standard libraries and whether Google was allowed to use the same or similar APIs in Android even though the implementation of those interfaces was different. Most technology people believe that APIs should *not* be copyrightable since they are functional, and so that programmers can use consistent and familiar ways to interface to different platforms. This case has gone all the way up to the US Supreme Court, which ruled narrowly for Google, but did not resolve the issue in general.

Often you will see copyrights indicated by having the copyright symbol © somewhere near the front or at the bottom of a work. However, copyrights are available free and automatically for all works, even if the © symbol is not present. Authors may register their work, for example with the US Library of Congress, to help secure a clear date of authorship, but that is optional.

22.3 Trademarks

A *trademark* is a distinctive phrase or indicator that uniquely identifies a particular commercial product. The goal is to avoid confusion in the consumer's mind and to allow a company to protect their brand. Trademarks are only available for commercial products (and not, for example, for university research projects!). Unlike copyrights and patents, trademarks do not expire, as long as the product is available commercially. Trademarks can apply to a word, icon, color, and so on.

There were two important trademark issues identified in this book. The first (Section 3.13) was that the original Palm Pilot had to change their name to remove the word "Pilot" since the Pilot pen company had a trademark on the word "Pilot" and since the Palm has a stylus, which is sometimes called a "pen," this might be confusing. The subsequent versions were just called the "Palm" with a model name, like "Palm Professional" or "Palm V." Similarly, in a related case (Section 3.14), Microsoft tried to call their handheld device a "Palm PC," but were required to remove the word "Palm" since the Palm company had a trademark on that name.

Trademarks are indicated by a small symbol usually at the bottom right of the mark. The symbols include ™ for TradeMark and ® for Registered trademark, which is what one usually sees. Trademarks can be registered with the United States Patent and Trademark Office (USPTO). Other less common options are SM, for service mark, and wordmark, which is for a specific text-only font and presentation of a word. Registering and protecting a trademark requires a lawyer and is usually expensive.

22.4 Trade Secrets

A *trade secret* is any type of information that a company takes steps to prevent becoming known outside of the company and which has economic value. Of relevance to this book, an algorithm, for example, for quickly rendering 3D (Chapter 12) or for gesture recognition (Chapter 14), might be a trade secret that a company keeps hidden. However, most interaction techniques are pretty apparent from just seeing and using them (since they must be used and understood by users!), so trade secrets are usually not relevant to interaction techniques.

22.5 Patents

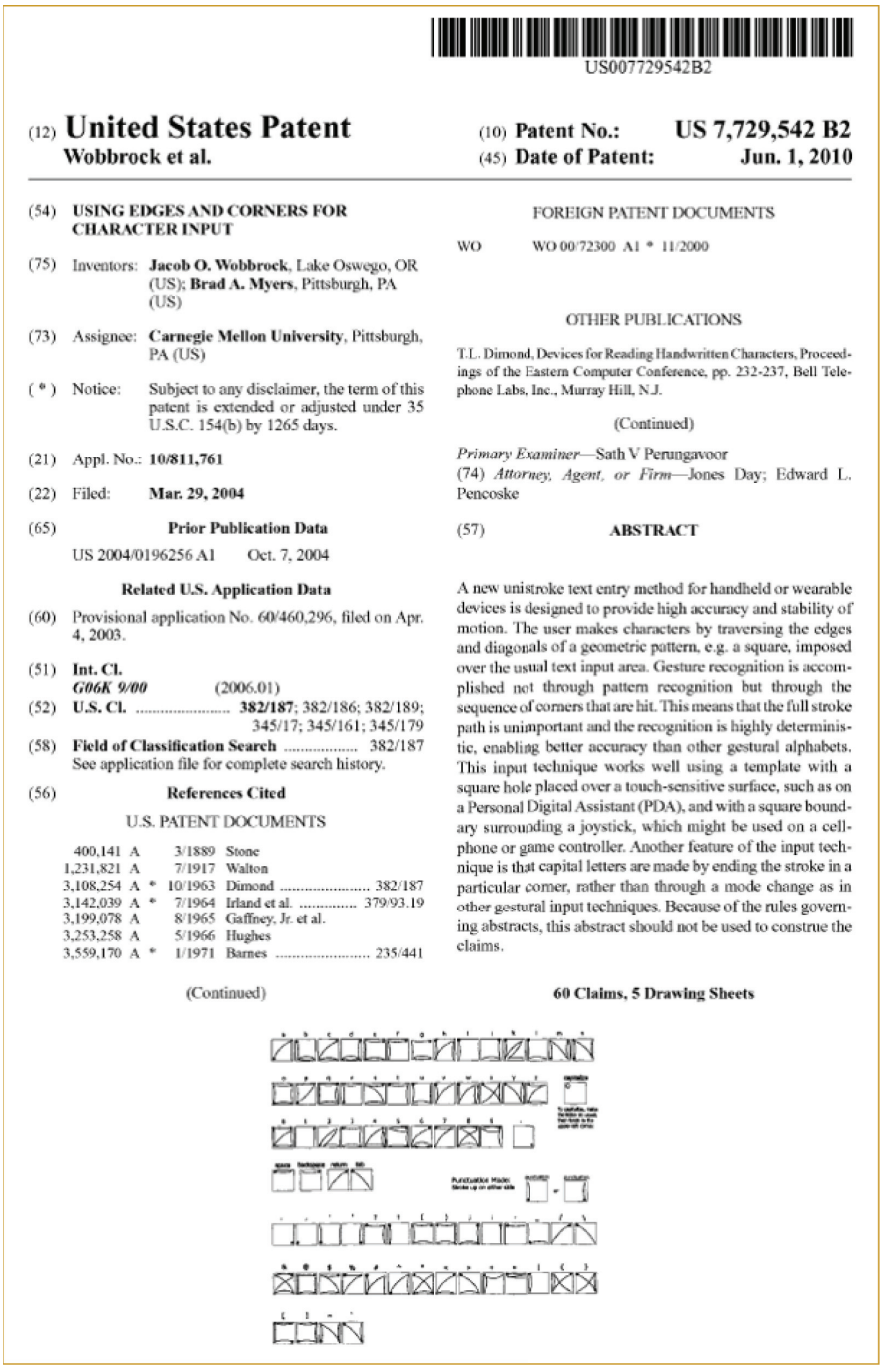
Patents are clearly the most relevant kind of intellectual property for interaction techniques. A patent gives the owner the right to prevent others from making or selling an invention for a fixed period of time (20 years in the US) in exchange for publicly describing how the invention works.

Originally, patents could only be issued for *physical* inventions, as articulated in many early court cases: "For an idea to be patentable it must have first taken physical form."[1]However, in 1981, the US Supreme Court decision in *Diamond v. Diehr* opened the way for patent protection for computer software.[2] Note that 1981 is *after* many of the interaction techniques described in this book were first shown, including menus, scrollbars, dialog boxes, icons, window managers, gesture recognition, and so on, so the early inventors did not try to get patents on them. That is why the early case of *Apple Computer, Inc. v. Microsoft Corporation and Hewlett-Packard Co.* from 1989 about copying Apple's look and feel was about copyrights and not patents (see Section 22.2).

In order to get a patent, you need to have a patent submission written up (usually by a patent lawyer) and submitted to the USPTO, who will evaluate the submission and usually return it for more work by you and your lawyer. This will often go a few rounds until the patent is revised enough to be acceptable, and then it is *issued*, at which point you can start trying to enforce your patent. This process can take many years (our patent on EdgeWrite [Section 8.3.2 and Figure 22.2] was seven years from first submission until it was issued). The lawyer and USPTO fees for getting a patent can easily run to USD $20,000, so unless you are at a big company it only pays to try to get patents on important inventions.

1. [Rubber-Tip Pencil Company v. Howard, 87 U.S. (20 Wall.) 498, 507 (1874); Clark Thread Co. v. Willimantic Linen Co., 140 U.S. 481, 489 (1891)]. *See:* http://caselaw.lp.findlaw.com/data/constitution/article01/39.html.

2. *See:* https://caselaw.findlaw.com/us-supreme-court/450/175.html.



US007729542B2

(12) **United States Patent**
Wobbrock et al.

(10) **Patent No.: US 7,729,542 B2**
(45) **Date of Patent: Jun. 1, 2010**

(54) **USING EDGES AND CORNERS FOR CHARACTER INPUT**

(75) Inventors: **Jacob O. Wobbrock**, Lake Oswego, OR (US); **Brad A. Myers**, Pittsburgh, PA (US)

(73) Assignee: **Carnegie Mellon University**, Pittsburgh, PA (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 1265 days.

(21) Appl. No.: **10/811,761**

(22) Filed: **Mar. 29, 2004**

(65) **Prior Publication Data**

US 2004/0196256 A1 Oct. 7, 2004

Related U.S. Application Data

(60) Provisional application No. 60/460,296, filed on Apr. 4, 2003.

(51) **Int. Cl.**
G06K 9/00 (2006.01)

(52) **U.S. Cl.** **382/187**; 382/186; 382/189; 345/17; 345/161; 345/179

(58) **Field of Classification Search** 382/187
See application file for complete search history.

(56) **References Cited**

U.S. PATENT DOCUMENTS

400,141	A		3/1889	Stone
1,231,821	A		7/1917	Walton
3,108,254	A	*	10/1963	Dimond 382/187
3,142,039	A	*	7/1964	Irland et al. 379/93.19
3,199,078	A		8/1965	Gaffney, Jr. et al.
3,253,258	A		5/1966	Hughes
3,559,170	A	*	1/1971	Barnes 235/441

(Continued)

FOREIGN PATENT DOCUMENTS

WO WO 00/72300 A1 * 11/2000

OTHER PUBLICATIONS

T.L. Dimond, Devices for Reading Handwritten Characters, Proceedings of the Eastern Computer Conference, pp. 232-237, Bell Telephone Labs, Inc., Murray Hill, N.J.

(Continued)

Primary Examiner—Sath V Perungavoor
(74) *Attorney, Agent, or Firm*—Jones Day; Edward L. Pencoske

(57) **ABSTRACT**

A new unistroke text entry method for handheld or wearable devices is designed to provide high accuracy and stability of motion. The user makes characters by traversing the edges and diagonals of a geometric pattern, e.g. a square, imposed over the usual text input area. Gesture recognition is accomplished not through pattern recognition but through the sequence of corners that are hit. This means that the full stroke path is unimportant and the recognition is highly deterministic, enabling better accuracy than other gestural alphabets. This input technique works well using a template with a square hole placed over a touch-sensitive surface, such as on a Personal Digital Assistant (PDA), and with a square boundary surrounding a joystick, which might be used on a cellphone or game controller. Another feature of the input technique is that capital letters are made by ending the stroke in a particular corner, rather than through a mode change as in other gestural input techniques. Because of the rules governing abstracts, this abstract should not be used to construe the claims.

60 Claims, 5 Drawing Sheets

Figure 22.2 Cover page of Jacob Wobbrock and my patent. Source: Cover page of US patent 7,729,542.

Patents have a number of parts. The cover page (see Figure 22.2) contains a collection of information about a patent, such as the title, inventors, patent number, abstract, when it was filed and issued, the references the patent officer considered, and a representative picture from the figures. Next is a collection of figures to illustrate the patent (in other venues besides the US, the figures may be at the end instead). The body of the patent is called the *specification*, which is required to explain how the invention works. Finally, the *claims* of the patent must describe the scope of the invention—what the patent specifically covers.
Patents are required to be:

- New (novel)—which means that you are only allowed to patent ideas that have not already appeared.
- Useful—it has some kind of commercial application, but this is a very low bar, and I have never seen it used as a way to dispute a patent.
- Non-obvious—it must not be an improvement that would be obvious to a person having ordinary skill in the pertinent art. This is often the key dispute with a patent.
- Properly disclosed—the invention can be reproduced by just reading the specification and figures.

Typically, when the patent office is considering a submitted patent, it is primarily looking for *prior art*—any publicly available publication that it believes discloses the same thing as the claims. The patent submitter's response is usually to make the claims more specific (narrower) so they only cover the novel parts of the invention and are thereby different from what the prior art discloses. For example, the first claim of the EdgeWrite patent of Figure 22.2 is:

(1) A character recognition method, comprising:

    - using a processor to perform the following method;
    - queuing continuous path data made by traversing edges and diagonals and into corners within a confining geometric shape constraining an input device to less than the input device's total area;
    - searching the queued continuous path data to generate a sequence of corner hits, where a corner hit corresponds to a corner defined by said geometric shape;
    - identifying a character based on said sequence of corner hits independently of the remainder of the continuous path data;

- wherein each of said sequences of corner hits defines a single stroke, and wherein each single stroke is representative of one of a letter, number, punctuation, or mode; and
- identifying a letter character as being upper case when said stroke representative of said character ends in a common predetermined corner and lower case when said stroke does not end in said common predetermined corner.

We originally tried to get a more general claim that would cover any method of corner hits to recognize characters, but the patent office objected, so we added the final clause about how EdgeWrite identifies uppercase letters (see Section 8.3.2), which the patent office agreed no one else had done before. So now, someone would have to do all parts of this claim, including the exact way of indicating uppercase letters, to be considered to infringe that claim.

In order to "prevent others from making or selling" a patent, you typically need to have lawyers to file a civil lawsuit in a court of law, and the defendant (who you are accusing) has the right to dispute it. This can result in legal fees that reach 1 million US dollars if it goes to a jury trial, so most cases are negotiated between the two sides. The defendant argues that their product does not do what the claims say, and furthermore that the patent office was incorrect in issuing the patent because of new prior art that makes the patent invalid because the patent is not novel and is obvious. The patent owner who brought the suit, called the *plaintiff*, argues that the product *does* do what the claims say and that the patent is *not* the same as what the prior art describes.

When a patent expires (in the US, 20 years after the filing date), the invention goes into the public domain, and anyone can use what is in the specification for free. Some companies will try to avoid this by getting new patents on the updated novel parts of newer versions of their products, but 20-year-old ideas are rarely worth much in the computer-related fields.

22.5.1 Patent Controversies

Since software patents were first allowed in 1981, the number has exploded (see Figure 22.3), and many of these can be considered patents on user interfaces and interaction techniques. However, software patents are controversial for many reasons. Some people are against any kind of software patents. For example, https://endsoftwarepatents.org/ argues that software patents are an impediment to innovation and an unjust threat to software developers as well as software users. An important issue is that the distinction between an unpatentable *abstract idea* and a patentable implementation of that idea in code is not clear, and a variety of

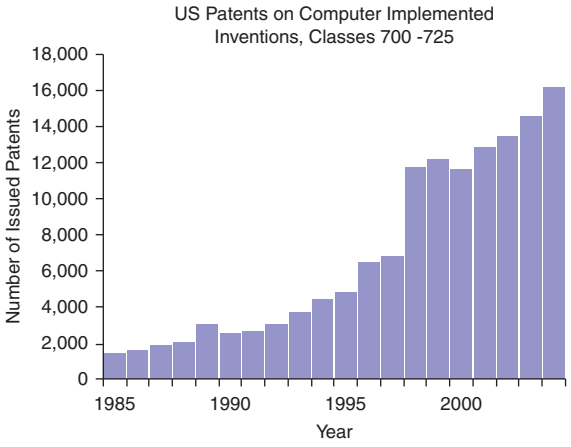


Figure 22.3 Number of software patents issued. Source: By Nowa, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=2682254.

court cases, such as the famous *Alice Corp. v. CLS Bank* case (2014) that was decided by the US Supreme Court, have not definitively cleared this up. Further, a modern piece of software or device (like a smartphone) has thousands or millions of elements in software and in the user interface (and dozens to hundreds of interaction techniques), so assigning a *value* to a single patented element can be difficult.

Another consideration is the sheer number of software patents that have been issued and the vast number of lawsuits about them. The rise of companies whose entire business is about collecting patents and suing companies, called a non-practicing entity (NPE) or a patent assertion entity (PAE) or pejoratively, a *Patent Troll*, has exacerbated this problem. (NPE is actually a broader term since it includes universities that own and assert patents.) Now, almost any business of any size might be threatened with a lawsuit from one of these entities.

On the other hand, as an inventor of interaction techniques myself, I believe that we should be able to protect and profit from our inventions in the same way that an electrical engineer or chemist can, and we shouldn't be penalized because our inventions happen to be in software instead of in hardware or chemicals.

22.6 Considerations for Intellectual Property and Chapter Conclusions

There is much more that can be said about intellectual property with respect to interaction techniques and user interfaces in general. Hopefully, this chapter has given an overview of the important terms and issues. If you are a company *using* interaction techniques, the basic interactions in the libraries are probably safe, but

you still might be sued for how your application puts them together or how they are used together. If you *invent* a new interaction technique, it would probably be appropriate to try to get a patent on it, if it has market value. Many big companies, like IBM, Microsoft, and Apple, will file for software patents on many of their software inventions, partially as a deterrent in case they are sued by another company. Even if you want to put your technique into the public domain, as Loren Brichter did with the pull-to-refresh interaction technique (Chapter 32), it is useful to have a patent to make sure no one else tries to patent it first.

On the other hand, getting a patent can be quite expensive, so smaller entities, like university researchers and startups, may need to be strategic about only filing patents for inventions that seem to have commercial value. Of course, companies should trademark their significant brand identity and logos and consider whether to have trade secrets and copyrights on their text and graphical expressions.

Given the significant value of intellectual property, but also the significant costs to companies of lawsuits about infringing intellectual property, and the significant ambiguity about what can be covered, one can hope that government and courts will be able to provide more clarity in the future.

PART IV

INTERVIEWS

This part includes summaries of the guest lectures by inventors of some important interaction techniques and interaction paradigms. These were given in my course on *Interaction Techniques* that I have given at Carnegie Mellon University four times so far (as of summer 2022). Four of the summaries are based on a class project where a team of students summarized those lectures.

The guest lectures are a popular part of the course each year. Some of the inventors are my personal friends and colleagues, and others I just contacted through LinkedIn and other connections. It is amazing to think about how something as fundamental as the copy-cut-paste or the pull-to-refresh interaction technique was invented by an individual (Larry Tesler and Loren Brichter) who can speak to the class about it! Other speakers were involved in whole applications like the invention of spreadsheets (Dan Bricklin) or the desktop (David Canfield Smith).

A noticeable absence is people from Apple who can talk about all of their innovations and interaction techniques, beyond the original Macintosh. Unfortunately, Apple's culture and policies seem to prevent anyone from there commenting on how their interaction techniques come to be, even though the people involved and descriptions of the behaviors are often well-described in patents (e.g., Ording [2008, 21:27–30], Guglielmo [2012], and Williams [2013]). Other people I would like to have included have passed away (like Mark Weiser for UbiComp and Jean-Marie Hullot for Interface Builders). Many others declined to participate or declined to allow their lecture to be recorded, or who I haven't gotten around to inviting yet.

I expect to continue teaching this course, and I will be inviting other inventors in the future. You might check the website for the book (ixtbook.com) to see if there are new summaries since this edition of the book was published. If you know of anyone who would be good to have speak in a future iteration of the course (or if you would like to volunteer), please send me the contact information!

The summaries are presented in approximately chronological order of the key contribution they are known for.

CHAPTER 23

Interview: Larry Tesler

Figure 23.1 Brad Myers giving Larry Tesler the "SIGCHI Lifetime Practice Award" in 2011. Source: Picture by Ben Shneiderman; reproduced with permission.

Guest lecture from February 5, 2014. This chapter is based on a summary created by Jae Kim, Jason Hsu, Tianying Lyu, and Simon Corpuz for class in Spring 2022, reused with permission. See also Tesler's various retrospective articles (e.g., Tesler [2011, 2012]).

23.1 Short Bio of Larry Tesler

Larry Tesler (Figure 23.1) was one of the most influential computer scientists and designers during the early days of commercialized computing. Many of his ideas, contributions, and discoveries have shaped the course of the personal computer's evolution, and they are still evident in modern computers. He was closely involved in the invention of many of the now-familiar low-level interaction techniques in all graphical user interfaces. These included cut-copy-and-paste editing along with their keyboard shortcuts, inserting and overwriting text without entering a mode by simply clicking or dragging and then typing, typing or pasting find and replace text into a form that can be edited before and after searching, between-character text insertion points, drop-down menus, paned-window browsing (later called frames in web browsers), and the GUI integrated development environment (IDE) for programmers including code browsers, object inspectors, and stack-inspection debuggers. He joined Xerox PARC in 1973 and transferred to Apple in 1980 where he was a key leader of the Apple Lisa and Macintosh designs. Larry Tesler passed away on February 16, 2020.

23.2 User Studies—Stanford Card Stunt Program

Larry Tesler was first introduced to the concept of a user study during his time in college when he was in charge of the Stanford Card Stunt Program from 1962–1964. Card stunts are a performance by students during football game halftimes, where each individual student was given a set of instructions telling them which color card to raise to form certain graphics (Figure 23.2). Before the existence of the Card Stunt Program, artists and organizers had to hand-design the graphics (some are even animated) as well as manually fill out instruction cards for each student. As the card stunts became larger and more elaborate, the students eventually decided that this process needed to be automated—hence the origin of the Card Stunt Program. The software allowed artists to digitally create their designs using computer graphics, and it then could automatically generate instruction cards that incorporated all the graphics used in the show, for every student.

However, the initial version of the program was difficult for the artists to use; the code that the artists had to write was basically "a long list of Xs and Ys," which caused the artists to make errors. The artists also thought it was okay to have typos and misspellings in the code since they didn't think those mistakes mattered. When Tesler was tasked with improving the program, he decided he needed to figure out what was wrong with it first. He quickly realized that the best way for him to discover issues in the program was a form of think-aloud study—quietly watching them work and encouraging them to verbalize their thoughts and concerns. It felt like the most "empirical and intuitive" way to figure out what was wrong in a

Figure 23.2 Stanford students lift appropriate cards to make a red and white letter "S." Source: Picture from ACM video: Tesler [2011].

user application. Through the observations he made with this technique, he realized the need for more descriptive error messages as well as better error checking during code compilation.

23.3 Blank Screen Study

After he started working at Xerox PARC in 1973, Tesler had the idea of a Blank Screen Study. Basically, he wanted to understand what was the most intuitive way users would like to interact with software, without any predisposing bias or constraint from working with existing software. He believed that some interaction techniques are "more intuitive than others," and he hypothesized that blank screen studies were a good method to extract those intuitive techniques from users.

The blank screen study Tesler conducted was for text editors. The user was provided with a document full of editing markup (deletions, insertions, substitutions, etc.), a keyboard, and a monitor that was not displaying anything. A mouse was avoided in this study to eliminate any predisposition users might have toward any certain type of pointing device, so fingers were used to simulate a pointing device. The user was then asked, "what would you do on the computer to do X?" and then the answer was observed.

One big discovery that came out of this study was that the operating syntax of computers at that time is actually not what people intuitively do. During the study, the user preferred a suffix (verb-after) syntax, meaning that the object that the user wants to perform an action on is selected first, before the action itself (e.g., delete, move, and rename). Computers those days, namely the NLS system, implemented a verb-first system, which had the users pick their actions before what items to perform them on (see also Section 10.2 for more about the syntax of operations).

23.4 RCG User Study

RCG (Research Character Generator) was a bare-bones version of the NLS text editor, without the graphics and networking. It originally supported a heavily-moded user flow where the syntax for most actions were prefix (verb-first). For example, if users wanted to delete text, they would have to press "D" first, choose what type of text they want to delete (character, word, statement, text), then select what to delete, and finally accept with a mouse click. Among users, opinions on RCG were polarized: some people felt that modes were too confusing to use and a downgrade from typewriters, while others believed that it brought a lot more text editing functionality to the table than traditional typewriters. To improve the usability of future text editors, RCG had to be evaluated for what it did well and what it did not. Some conclusions that Tesler made are:

- Character keys should only input characters, not characters and commands.
- The intuitive use of the mouse is to point at things, not to accept and delete commands.
- There should be a cursor displaying where the next typed character will appear.
- There should be a way to move the cursor without taking your hands off the keyboard.
- Invisible characters (whitespace characters) should not be commands.

The conclusions that Tesler and his colleagues reached have influenced text editor interactions to this day. They also signify the beginning of the shift from moded interactions in software toward unmoded interactions.

23.5 Cut and Paste

In 1973, Larry Tesler designed a cut-and-paste mechanism to be used in text editing. He drew inspiration from magazine editing, where sections of text and images could be cut out and arranged to form different layouts on a page. He also saw

Pentti Kanerva's text buffer demo, where he showed that he could move text from a document into a buffer and then unload the contents of the buffer into somewhere else in the document. The convenience of the buffer, Tesler noted, was that the user had the ability to perform other tasks between moving the text into the buffer and moving it out. With the use of a buffer, there wouldn't be a need for a copy/cut mode and a paste mode, which was tremendously useful as early UI designers were trying to move away from moded interactions.

When initially presented with the idea of cut and paste for text editing, Tesler's colleagues had a few objections, most notably being that cut and paste will be a foreign idea for most users, so it might become underutilized and perhaps even confusing to use. However, these voices were quickly cast aside as testing results showed that this idea of modeless cut and paste was much faster and less error-prone than the NLS's interactions, as well as providing users with the additional flexibility of performing other tasks between cutting and pasting— scrolling was especially important since it may not be possible to get the source and destination on the screen at the same time.

23.6 Modeless Software and the Gypsy Text Editor—The Shift Away from Moded Software

Tesler defines a *mode* as "a state of the user interface that lasts for a period of time, is not associated with any particular object, and has no role other than to place an interpretation of operator input." Typically, software that employs a prefix syntax is moded since the user must choose an operation first (e.g., delete) and then select the operand (e.g., characters, words). Early computer systems such as NLS were mostly designed with prefix syntax.

Back in the 1970s, mice were very inaccurate and often did not perform what the user intended it to perform, leading to very high error rates especially during high-precision operations such as selecting a word or character in a text editor. When an erroneous selection is made using RCG, which employed a prefix syntax, the user would have to cancel the operation and then reselect what they intend to select. Tesler thought that this process could be much more efficient with a suffix syntax text editor. In such a text editor, the user selects the character/word/passage before the operation is specified, so if a mis-selection is made, the user can just reselect without needing the additional action to cancel the operation.

Besides lower effectiveness, prefix syntax text editors also take up a higher cognitive load on its users, as in addition to the writing task users also must keep track of what mode they are in as well as what actions can be performed safely in this mode. Users also sometimes have trouble exiting the current mode as some modes

can cascade into other modes after a user action. In RCG, two types of exiting are permitted: "Backspace" brings the user up one mode in a cascaded interaction, and "Command Delete" will bring the user all the way out of the entire cascade. These were confusingly provided on the second and third mouse buttons.

23.7 The Gypsy Text Editor

The Gypsy text editor was created by Larry Tesler and his colleague, Tim Mott, in 1975. Gypsy was the first text editor to eliminate modes—instead of users needing to be in "INSERT" or "DELETE" mode, users can just press a character key any time and have a character typed into where the current cursor is, and the delete key will delete the character wherever the cursor is. In addition, Gypsy introduced a scan (search) function where the users could just click in the edit field to edit their existing search string, and that field also supported text selection with the mouse, modeless text replacement, copy, cut, and paste.

During the development of Gypsy, the concept of advanced shortcuts for power users was explored. For example, single clicking would select one character, double clicking (and drag) selected word(s), and clicks in the margin selected a paragraph. However, it was discovered that even after practice, users still found these power-user features hard to use and therefore seldom used them. The design was then reverted back to a simpler model where single click places the cursor between characters and clicking and dragging selected a character range.

Gypsy was developed using the insights Tesler made from observing earlier moded text editors such as NLS, and it received good reviews from users, highlighting the new modeless operating mode and how much easier Gypsy felt to use. The modeless movement quickly spread to new text editors and computer systems, many of them inheriting the new interaction techniques that were introduced in Gypsy.

23.8 Other Problems with Modes

Tesler identified a number of other problems with modes (and famously argued against them, see Figure 23.3):

> In Bravo, which was the editor at PARC just before Gypsy. ... E was select all, selected the entire document. D deleted whatever was selected. I entered insert mode, and T in insert mode just typed a T. So if you typed the word "Edit" thinking you were in insert mode, but you were actually in command mode, it would select the whole document, delete it, and then insert a T so that undo no longer worked, because we had a one-step undo. So, you had to be pretty careful when you typed the word "Edit," to make sure you were

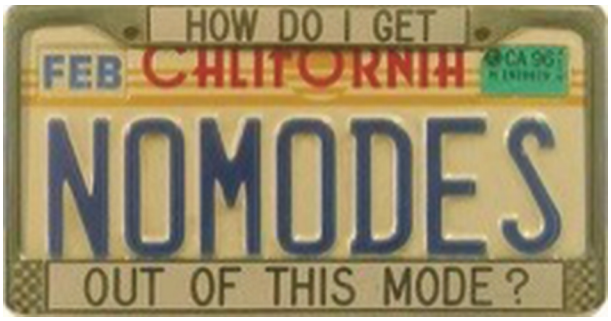


Figure 23.3 Larry Tesler's license plate was "NOMODES." Source: Picture by Larry Tesler, from https://www.nomodes.com/Larry_Tesler_Consulting/CV.html, reproduced with permission.

> in edit mode ... . So people realized pretty quickly that modes were bad. The other thing was that people would always say, how do I get out of this mode? And they felt stuck—and sometimes didn't know what to do. They didn't have a consistent mode escape, and some programs didn't have one.

23.9 Interaction Techniques at Apple

Tesler moved to Apple in 1980.

> I managed the Lisa user interface development. Since we had a license to Smalltalk ... and Smalltalk was public information. ... Dan Ingalls and I gave the demo to Steve Jobs back in 1979. So of course, we were not only familiar with Smalltalk, but we were allowed to use ideas from that. So that's all the cut paste copy, was taken directly from there, and undo and double-click and all that stuff. But other things that were innovations that I contributed to were the pulldown menu. I helped Bill Atkinson come up with that. In applications like Excel and Word, in the scrollbar, there's a split icon that you can drag and split your window into panes. Engelbart had actually come up with that, but he did that in a moded way, I came up with the idea of having that little icon in the scrollbar, at the end of the scrollbar, that you would drag. But mostly, I ran a lot of usability studies, and made recommendations about changes to designs that the programmers had come up with.
>
> The one button mouse. ... The very first Engelbart mouse was one button. ... In Gypsy we had the one-button subset that we defined, and in fact that was all that anybody ever used. ... And so when I ran my very first user test with the mouse, in addition to what mode am I in the other big question was what

button should I hit? And so I really wanted to get rid of the other buttons, it just took a while to figure out how to have just one button, that was—we figured, we finally figured it out in '75. But didn't even really do it. ... But we didn't get one-button mice until Apple. ... I wrote a memo with the help of Jef Raskin and Bill Atkinson about why we should ... [have a] one-button mouse, and the product marketing manager ... and Steve Jobs liked it, so we did that.

CHAPTER 24

Interview: David Canfield Smith

Figure 24.1 David Canfield Smith (right) giving a demonstration to Brad Myers (left) of the KidSim system [Smith et al. 1994] at CHI'1995, which eventually became Stagecast Creator. Source: Picture by Ben Shneiderman; reproduced with permission.

Guest lecture from February 3, 2014. This chapter is based on a summary created by Jae Kim, Jason Hsu, Tianying Lyu, and Simon Corpuz for class in Spring 2022, reused with permission. See also a similar lecture by Smith called "The story of icons" from 2021 on YouTube: https://youtu.be/Bt_zpqlgN0M.

24.1 Short Bio of David Canfield Smith

David Canfield Smith (Figure 24.1) coined the term "icons" in his landmark 1975 Ph.D. thesis *Pygmalion: A Computer Program to Model and Stimulate Creative Thought*. He was a leader of the user interface design team for the Xerox Star, which innovated many important interaction techniques that are still used today and others that are arguably more effective than what we currently use. David was also a leader in the area of programming by example.

24.2 The Story of Icons

David Canfield Smith first defined an icon (see also Sections 2.4.6 and 11.6) in his 1975 Stanford Ph.D. thesis on Pygmalion as a small picture on a computer that represents some kind of resource. Smith said that an icon has two important aspects: its visual semantics and its machine semantics. Visual semantics means how the icon looks to users, but machine semantics are a bit more complex. They comprise the icon's various features, such as its properties or attributes, the content of its functions, and any working code. In other words, the icon can also provide the proper functionality of the resource that it represents. For example, an icon of a printer should be able to initiate the process of printing documents when interacted with.

Smith recalled the way the name "icon" came about:

> So one day, Alan Kay and I were sitting around in his office, discussing the progress of the thesis, and he mentioned that we need a name for these objects that users are manipulating on the screen, on the so-called blackboard. And we rejected names like picture, image, sketch, painting, drawing ... because they didn't capture the idea that the objects have both visual semantics and machine semantics. They're not just bits on the screen. And Alan pointed out that in certain religions, such as the [Eastern] Orthodox Church ... images portraying holy figures are believed to be holy objects in themselves. So, a picture of the Virgin Mary is thought to have some of the holiness of the mother of Jesus. That is to say, they embody some of the attributes, or semantics, of the thing they are portraying. That's perfect. The Orthodox Church's name for these images is "icon."
>
> I also considered things like "totem" because that's also considered to have mystical properties, but that comes with a little too much conceptual baggage, so I rejected it. Otherwise, we might be manipulating totems when we use a PC. Instead, we settled on icon. So, the key idea is that the picture is more than just bits on the screen or pigments on a canvas.

Icons were used in two of Smith's important projects, including the executable blackboard Pygmalion, which was his dissertation work, and the Xerox Star software. It is thanks to these systems that icons became popularized across the market for interaction technique design.

24.3 Pygmalion

Smith has stated that during the process of invention we usually take small steps. However, there is usually an initial "aha" moment of inspiration before the real work can begin. For Smith, that flash of inspiration came to him during his college years. While mapping out a visual diagram of code on a blackboard, he noted the difficulties that would come from translating the material from the blackboard into the actual programming language. From this problem, he developed a unique solution for its time: run the diagram itself, by giving it the functionality of code.

Smith's 1975 dissertation revolved around the creation of Pygmalion [Smith 1977]—an executable blackboard that allows users to create diagrams using icons and then directly execute those diagrams as working coded functions. These icons are directly drawn onto the interface in a blackboard-like window that takes up most of the screen. Users can then utilize the pre-existing operations in the menu window, along with clarifying any details within the other boxes along the bottom of the screen, to help define the functionality of each icon (see Figure 24.2). Pygmalion as a whole was implemented using the Smalltalk programming language (Section 2.4.3.1) on the Xerox Alto computer (Section 2.4.3), which had just recently been created by Xerox PARC.

In addition to being the starting point for icons, Pygmalion introduced the concept of programming by example (Section 15.13), also known as programming by demonstration. When users create a function in Pygmalion, they define various states that the function can be in. The system remembers each of these states along with the results that come from them, meaning they can be used as a part of other calculations later down the line. Smith demonstrated this process during his lecture by recreating the factorial function in a step-by-step format.

24.4 Xerox Star

In 1976, one year after creating Pygmalion, Smith would move on to work at Xerox as a developer of the Xerox Star. The Star was intended to be a consistent, user-friendly interface for office workers, at a time when computers were rarely used in commercial business. Smith wanted to reuse the icons from Pygmalion to help with making the interface more familiar to its intended users. The visual semantics of these icons (a printer, a calculator, folders, etc.) created an environment as close to an actual office space as possible on the Star (see Figure 2.14).

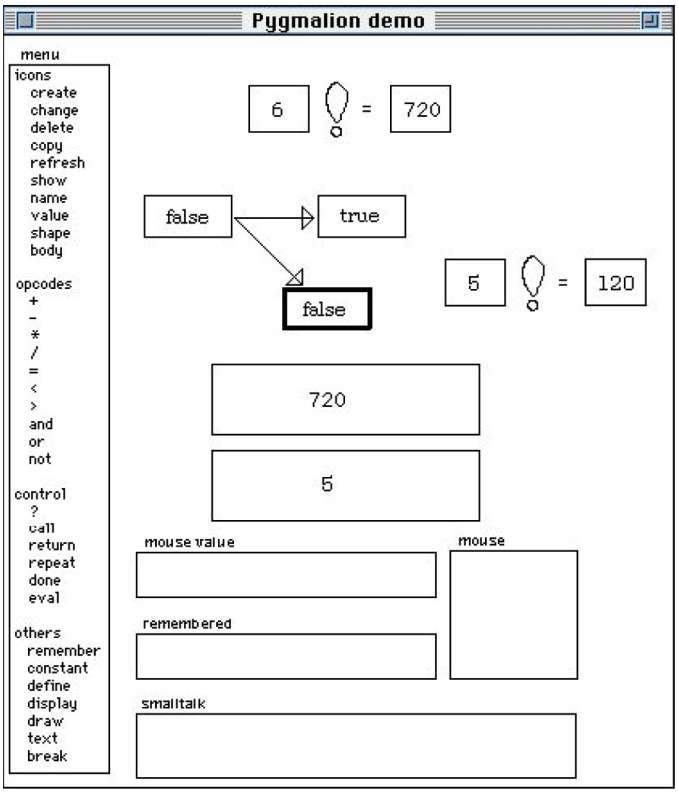


Figure 24.2 Final screen from demonstration of factorial using Pygmalion. Source: Reproduced from Chap. 1 of [Cypher et al. 1993] with permission.

This type of office-like environment created by the Star would come to be known as the "desktop metaphor" (see Chapter 11). Smith stated that two vital aspects of the desktop metaphor are power and familiarity. Basing the interface on an office space would allow workers to easily adjust to the new layout. And while many people believed that bringing the office to a digital setting would cause more limitations compared to the physical version, it actually opens up more opportunities to augment resources and give the users more power. For example, they added operations like sorting and filtering to the folders in Star, even though those are not available for a physical folder. They felt that using the desktop metaphor would allow them "to bring the office into the computer, not the computer into the office" and have "people to be able to use their existing knowledge about how to do things in an office, in order to understand this new system that was going to be thrown on them." The conventional wisdom at the time was that "executives don't type. That didn't turn out to be true."

Smith was specifically *not* assigned the task to design the *look* of the icons, which was assigned to a professional graphic artist (see Chapter 26). Smith *was* in charge of determining the machine *semantics* (meaning) of the icons. He identified two types of icons present in the Xerox Star. The first was data icons, which were multimedia documents, folders, and databases created by the user. The second type was function icons, which were provided by the system and acted like their real-world counterparts: filing cabinets, in- and out-baskets for mail, printers, wastebaskets, and so on. Specifically, function icons are interacted with by dragging and dropping data icons onto them. When this happens, the function icon performs its action on the data icon. For example, dragging a document icon onto the printer icon will begin the process of printing that document. Another example would be dragging a document icon onto the outbox icon, which queues that document to be sent via email.

24.5 Techniques for Reducing System Commands

During his lecture, Smith expressed how proud he was of the work done on the Xerox Star—specifically with regard to how efficient the interface was with its system commands. Whereas many modern systems such as Microsoft Word have numerous commands that often clutter the system, the Star was implemented in such a way that only a handful of commands were ever necessary at one time. Smith cited three methods that overall reduced the number of system commands in the Xerox Star: universal commands, creation by copying, and property sheets.

24.5.1 Universal Commands

The Star has a handful of system commands that are available no matter what application is currently open. Referred to as "universal commands," their utility encompasses many operations that other systems would have extra commands for. Smith listed off eight universal commands from the Xerox Star's keyboard (see Figures 2.14, 9.1, and 15.1):

- Move
- Copy
- Delete
- Undo
- Again
- Show Properties
- Copy Properties
- Help

Universal commands were chosen by going back to first principles and examining the fundamental computer actions that underlaid each command in an application. For example, internally, a great number of commands have the effect of moving something, which was abstracted into a universal Move command. People could use this one command to move a word, a sentence, a paragraph, a line in a drawing, a bar in a bar chart, a row in a table, a term in an equation, an icon on the Desktop, or to perform drag-and-drop. Smith estimated that the Move command in particular and universal commands in general reduced the number of commands in the system by an order of magnitude, without eliminating any functionality.

24.5.2 Creation by Copying

People often have a difficult time creating something out of nothing. They find it easier to modify an existing thing. Star elevated this observation into a fundamental way of working: creation by copying. The universal Copy command was designed to make this easy. In many cases, people could create a new thing by selecting an existing one and pushing the Copy key. Then the creation task became the easier one of editing something. The copied thing could either be supplied by the system, such as a generic bar or pie chart, or created by users, such as a form to be filled in. Creation by copying is a powerful idea that is drastically underused.

24.5.3 Property Sheets

Property sheets are windows that display the attributes of a data icon and allow users to easily edit those attributes. For example, the attributes for a text character might be its font, font size, emphasis (bold, italic, underlined), superscripted or subscripted, and so on. For a graphic line they might be thickness, color, solid or dashed, curved or straight, and so on. All the choices were shown in the selected object's property sheet—relieving users from having to remember them. Any attribute could be changed just by clicking on a different value. Users needed to learn only *one* technique to modify anything, which replaced hundreds of other commands. Smith referred to this concept as the "rule of two," where "two options is more than twice as complicated as one" since the user now must make a choice.

Apple used the term *dialog boxes*, which Smith thought was the better name. Smith said that previous systems at Xerox PARC had fixed areas for properties, but he "invented the idea of a little window that could be brought up on any object, or any selection. ... You selected something, and you push a little Properties key, and up would come a dialog box. So, in the ultimate reduction case, where I've selected a character and hit Properties, up would come properties ... . But then, there were hundreds of other property sheets that we also implemented in Star. I basically generalized the idea into a paradigm."

24.6 Other Star Designs

Smith was asked about the arrows on the Star scrollbar and how they point inwards instead of outwards like Apple's (see Figures 5.10(b) and 5.11). He said:

> That came directly from user testing ... , and so we implemented two ... prototypes. One of the scrollbars pointed the way the Mac does it, and one with the scroll arrows in the other direction. And it was pretty much unanimous among the novices that we tested that they liked the Star way. The Star way was, if you wanted the content of the window to move upwards, you push the up arrow. It made the contents move upwards. The Mac is based on the idea that going upwards, pushing the up arrow, makes the window contents go downward, because you want to go upwards in scroll, to an earlier point in the scroll. Well I'm sorry very much, but ordinary people don't think that way—they don't think in terms of scrolls. It's much more simple and concrete than that.

Similarly for selecting text, user testing showed that some users had trouble drawing through text to be selected holding a mouse button down, particularly if the text was on more than one line, crossed a page break, or required scrolling. So, Star used a two-button mouse, one to begin a selection and one to end it. Smith argued that "the Macintosh also used a two-button mouse, only the second button is on the keyboard—the Shift key!"

CHAPTER 25

Interview: Bill Buxton

Figure 25.1 Bill Buxton (left) with Brad Myers at the 2014 SIGCHI conference award banquet. Source: Photo credit: Ben Shneiderman, reprinted with permission.

Guest lecture from February 15, 2022.

25.1 Short Bio of Bill Buxton

William A. S. (Bill) Buxton (https://billbuxton.com/) (Figure 25.1) is a relentless advocate for innovation, design, and, especially, the appropriate consideration of human values, capacity, and culture in the conception, implementation, and use

of new products and technologies. This is reflected in his research, teaching, talks, and writing—including his past column on design and innovation for BusinessWeek.com and his 2007 book *Sketching User Experiences* [Buxton 2007]. He is now quasi retired, primarily devoting his time to curating his extensive collection of interactive devices, and other passions – especially his extended family. Beginning in December 2005, he spent 17 years at Microsoft Research as a Partner Researcher. Prior to that he was on the faculty in the Computer Science Department at the University of Toronto, where he continues to have an adjunct professor appointment. From 1987–1989, Buxton was in Cambridge, England, helping establish a new satellite of Xerox's Palo Alto Research Center (EuroPARC). From 1989–1994 he split his time between Toronto, where he was scientific director of the Ontario Telepresence Project, and Palo Alto, CA, where he was a consulting researcher at Xerox PARC. From 1994 until December 2002, he was chief scientist of Alias|Wavefront (now part of Autodesk) and, from 1995, its parent company SGI Inc.

He was also my advisor when I got my Ph.D. from the University of Toronto.

25.2 Interacting with Two Hands

Bill Buxton started with computers to make computer music, and as a musician, he was used to playing instruments with two hands. Using a stick-shift in a conventional car also shows how well people can coordinate multiple complex actions that have to be done at the same time. Similarly, when people write with a pen, the non-dominant hand is used to move the paper into a comfortable position. This was explained by Guiard's Kinematic chain model of bimanual asymmetric motor action, which states that the non-dominant hand moves first and sets the location of the action, and then the dominant hand does the fine motor action [Guiard 1987].

The early systems, like on the TX-2 (Section 2.4.1), did support two-handed interaction, but the non-dominant hand was only used to press buttons. Buxton then explained the two-handed studies that he and I did together in 1985 [Buxton and Myers 1986]. In the first study, participants had to move a box with the dominant hand and change its size with the non-dominant hand. We trained participants with the actions performed separately, and all but one of the participants naturally used both hands together in parallel. The second study compared scrolling with the non-dominant hand and clicking on things with the dominant hand (see Section 5.11.1 and Figure 5.24) versus using a regular scrollbar for scrolling. The results were that two-handed performance was 15% better for expert users and 25% better for novices. The scrollbars used in the study were conventional design from the Macintosh, and Buxton emphasized that they are composed of multiple sub-actions (dragging the elevator, pressing and holding the buttons, etc.), as discussed above in Section 5.6.5.

25.3 Adjusting Values

In 1978, Buxton demonstrated a music system with a character terminal connected to a pointing device using a graphics tablet (Figure 25.2), where if you pointed to a numerical value, pressed a mouse button, and pulled down or up, it would adjust the number as if it was a potentiometer. Furthermore, if you moved sideways, it would use a Vernier effect so users can get coarse and fine-grain control. Having physical controls at known positions, like the physical sliders in Figure 25.2, allows users to acquire and adjust the controls using motor memory without looking.

25.4 Future

For the future, Buxton argues that the palettes and menus should be manipulated with the non-dominant hand, like a physical painter's palette, rather than being around the active area because these don't work if you go to big or small displays.

> The UI has to come to you. It has to be in-hand, and disappear the minute that you can't use it. ... The menu, the iconography is exactly the same as what you've used for years there's nothing to learn except how bad the previous designs have been relative to what they could have been. Your job is to be frustrated with that, and not accept it, and come up with a better way, and then make it possible.

Figure 25.2 Music system from 1978. From left—physical buttons and sliders, character terminal with columns of numbers that can be selected and set using the keyboard or pointer; piano keyboard; digitizer with puck pointing device with four colored buttons. Source: Still from video Buxton presented, reprinted with permission.

CHAPTER 26

Interview: Norm Cox

Figure 26.1 Norm Cox. Source: Norm Cox, reprinted with permission.

Guest lecture from February 1, 2016. This chapter is based on a summary created by Jae Kim, Jason Hsu, Tianying Lyu, and Simon Corpuz for class in Spring 2022, reused with permission.

26.1 Short Bio of Norm Cox

Norm Cox (Figure 26.1) attended Louisiana State University to study Architecture and Design before beginning his career as a designer at Xerox in 1972, where he designed typefaces for the company's typewriters. He then joined Xerox PARC as a visual and interface designer where he designed the now widely used "hamburger" icon (three horizontal lines), which when pressed pops up a menu (Figures 5.10(b) and 7.2(a)). In 1982, he co-founded his own interaction and design consultancy, Cox & Hall, providing UI/UX design and visual design services. Cox is named

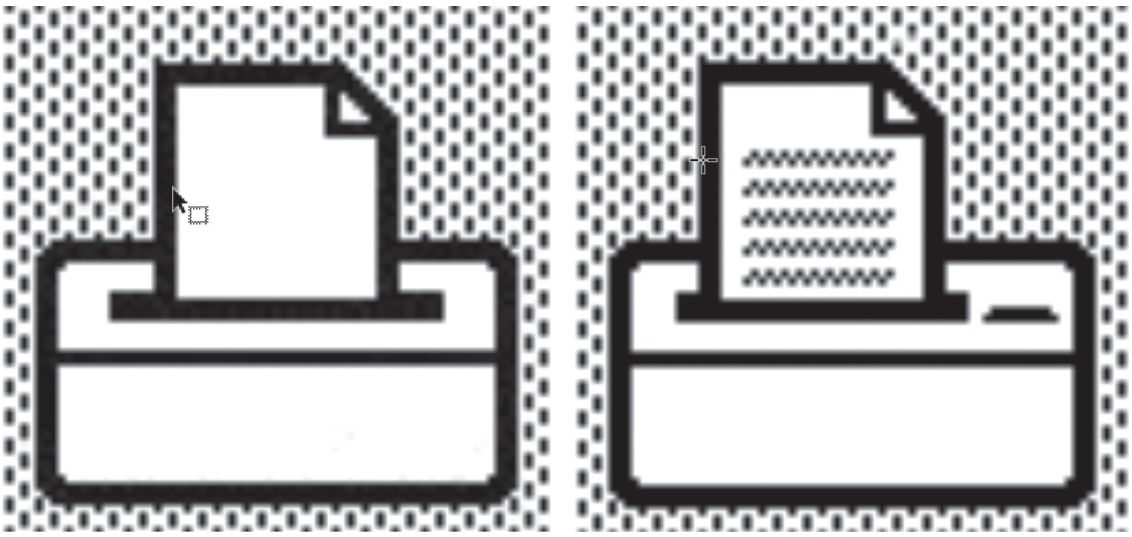
as inventor on 29 design patents for Xerox, IBM, Sun Microsystems, and Samsung. He has also received numerous industry design awards, juried international design competitions, and is a frequent lecturer at conferences, companies, and universities.

26.2 Visual Icon Design

Cox was tasked with designing icons for the Xerox Star computer. These icons were designed to be displayed on a Star desktop that was on a 17" black and white CRT display, and the icons had to be 64 × 64 pixels. His intention was to create icons that resemble the real objects that they are representing such as mailboxes, paper files, and printers. To design the most communicative and intuitive icons, he referred to several sources for inspiration. He looked at traffic signage, which places emphasis on shapes and relies on simplicity of images. Cartoon and cave paintings were also sources of inspiration for Cox in which objects are represented with minimal use of lines. He also looked at mechanical drawings or renderings that have purposeful line weights and visual hierarchy. His design tenets for the Star icons were simplicity, emphasis on shapes, consistent visual style, concrete (rather than abstract) images, and the metaphor of an office. But the most important tenet at the time was that icons needed to be user friendly—icons had to be familiar and easily recognizable to any viewer looking at them.

To evaluate how well people understood the representations of these icons, Cox and his colleagues conducted usability studies, which were some of the first user studies conducted on graphical user interfaces. His team had several methods to test icons, including a *naming test*, *timed response test*, *semantic differential test*, and *personal preference*. In a *naming test*, they showed the icons to participants and asked the participants to name them. For example, the original design for a printer icon (Figure 26.2(a)) elicited names like "shredder," "tissue box," and "toilet paper dispenser," so it was redesigned (Figure 26.2(b)), which worked somewhat better. The *timed response test* evaluated the time it took a participant to pick an icon from a group of other icons. The *semantic differential test* required participants to rate their attitude by selecting a position on an adjective scale. Cox recounts that one of the biggest lessons learned from these user studies was that users will always find ways to misinterpret an icon, and it is the designer's job to communicate meaning as effectively and efficiently as possible.

One of the most widely used icons today that Cox had designed is the *hamburger menu icon* (Figures 5.10(b) and 7.2(a)). When asked about the history behind this icon, Cox recounted that this was just one of many icons he designed at Xerox and that he did not expect it to be as impactful as it turned out to be at the time. The



(a) (b)

Figure 26.2 (a) Original and (b) final design for the printer icon. Source: Norm Cox, reprinted with permission.

original hamburger menu, invented for the Xerox Star, was meant to serve as a container for all the buttons that did not have a place on the header of a window. In other words, when the window header was full of buttons placed horizontally, the remaining series of buttons would be automatically placed inside the hamburger menu.

26.3 Legal Consulting

Norm Cox was asked by the lawyers for Microsoft and Hewlett-Packard to be one of their expert witnesses in the original lawsuit of *Apple Computer, Inc. v. Microsoft Corp. and Hewlett-Packard Co.* (see Sections 22.2 and 22.5). Originally, Cox said no since both companies were his clients, but he was convinced when the lawyers told him that "Apple is claiming that they own all the work that you did at Xerox." To which Cox said, "when do we start?" Since patents were not available at the time, Apple relied on Copyrights and was trying to claim that because they enhanced the original ideas from Xerox, Apple could prevent others from using it, but the judge ruled that Apple could only copyright what was actually unique to the Apple graphical interface. Cox said:

> So Apple came back with 188 items in their list of things that they claimed they improved the interface with, and over the next 4 years we went through

each of them systematically and determined that they were either in the public domain prior, or were owned by Xerox, or were designed by someone else. The judge explained that you can take an existing novel and add a chapter, or an existing movie and add a scene, but that doesn't mean that you can now claim ownership of the entire book or movie just because you added to it. For example, you can write another chapter to "War and Peace," but you don't now own the rights to the whole book. You only own the rights to what you added to the entity. So, after reviewing each of the 188 items in their list, Apple was left with ownership of only 2 of them. One was the trash can, and the other was expanding windows to open and close a window.

26.4 Interaction Design for Scrolling

Cox also worked on designs for navigating multi-page documents. At the time of his work for the Xerox Star computer, the feature to navigate between pages via a scrollbar as we know it today did not yet exist, so his team had to design it. He considered designs for the increment buttons at the ends that either pointed in or out. They asked participants what they would *expect* to happen when you clicked on an arrow at the top that pointed up, and they said the text would go up, so the Star used arrows that pointed inward (i.e., down at the top), since that is the direction that the text would move.

Cox also experimented with variations to the scrollbar that later influenced other designers in the design community. For example, during his time working for Metaphor Computer Systems, Cox replaced the vertical scrollbar with one simple icon that functioned like a joystick and allowed users to easily navigate up, down, left, and right on a window (what we now call *rate-controlled scrolling*, Section 5.8). Such simplicity in design was compatible with the Metaphor Computer System that itself aimed for simplicity and was inspired by industrial design.

Cox is also credited for designing a variation to a scrollbar for Sun Microsystems OpenLook in which the up and down arrows were attached to the thumb of the scrollbar instead of being positioned at the top or the bottom-end of the page (see Figure 5.14(a)). This prevented users from having to move the cursor to the ends of the window. Interestingly, this design resurfaced in 2010 when Google used a similar design for its scrollbar on the Google Wave software.

26.5 Mnemonic Gestures for Interacting with a Device

Cox worked with Samsung in 2014 to design new gestures for its mobile phone screens. He asserted that many of the finger gestures on screen displays including tapping, double tapping, pinching, and flicking do not have any mapping to

real-world gestures. Therefore, users are prone to forgetting which gestures trigger which event and are confused when a gesture does not work as anticipated.

When an engineer from the company approached him that Samsung has developed an on-device diagnostic feature that runs tests to troubleshoot issues on the device and improves its performance, Cox suggested using a gesture that is used in the real world—taking the pulse with two fingers on your wrist. Cox found out that using a mnemonic gesture that mimics a real gesture commonly used in real life led to users not forgetting how to invoke the diagnostic feature on the phone.

CHAPTER 27

Interview: Dan Bricklin

Figure 27.1 Dan Bricklin. Source: Dan Bricklin, reprinted with permission.

Guest lecture from February 20, 2019.

27.1 Short Bio of Dan Bricklin

Dan Bricklin (Figure 27.1) is the co-creator, with Bob Frankston, of VisiCalc, which was the first spreadsheet program, in 1979. He also founded Software Garden Inc. in 1985, which created "Dan Bricklin's Demo Program," which was one of the first programs to enable the layout and prototyping of UIs by drawing them. His book, *Bricklin on Technology*, was published by Wiley in May 2009.

27.2 VisiCalc

Dan Bricklin related the history of the design of the original spreadsheet, which was eventually called VisiCalc. The original idea was to have an AR heads-up display

with an input controller the size of a calculator augmented with a mouse hardware on the back, but they soon moved to having it work on an Apple II. The key goal was to make sure that the UI would be easier, or at least not harder, than doing the same thing on paper with a calculator. Figure 27.2 shows an initial plan from Fall 1978 for the keystrokes to invoke various modes and functions. Although Bricklin wanted to use a mouse to control the cursor of the application, the Apple II didn't have one, and the controllers it did have for games were too inaccurate and clunky, so he switched to using the Apple II's two arrow keys, which had to switch modes from horizontal to vertical movements using another key. The original version used heuristics to differentiate formulas from strings (so the "=" was not originally needed before formulas that Bricklin said came in with Excel), and it automatically detected and formatted labels differently from numbers. Identifying a cell using a letter and number, like C2, was already used in that version, and Bricklin explained that the letter had to go first to be clear that it was a cell reference (which would be ambiguous with 3+2C instead of 3+C2). One could reference a cell

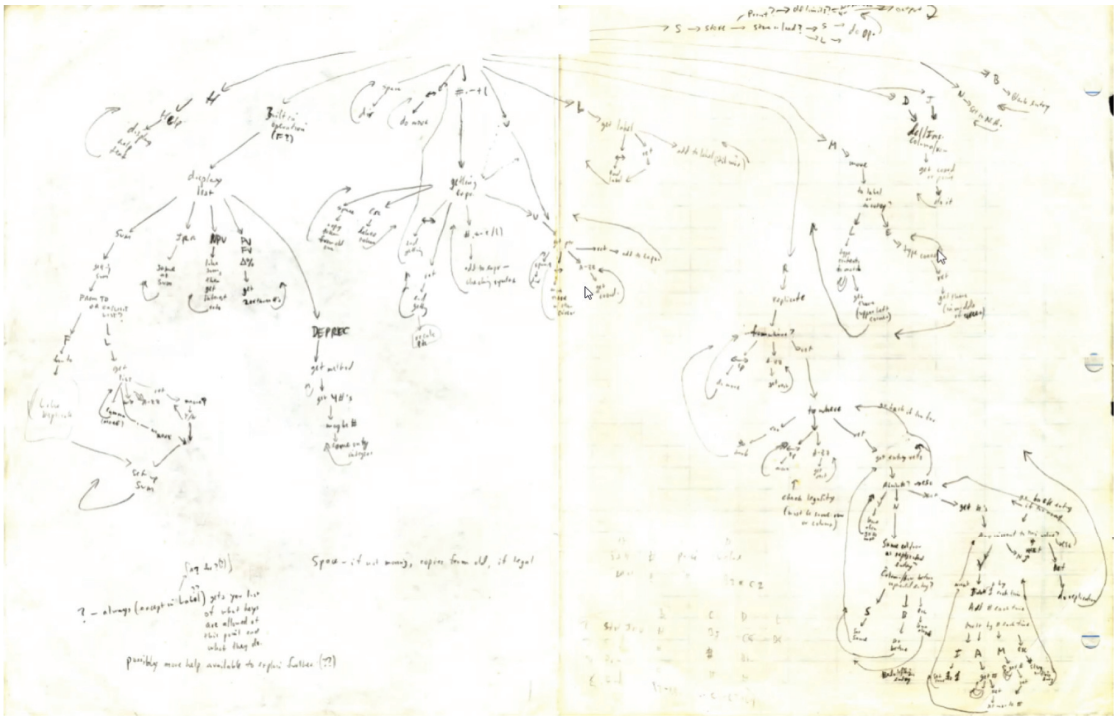


Figure 27.2 Early plan (Fall 1978) for the operations and modes of the spreadsheet, and keystrokes to invoke them. Source: Dan Bricklin, reprinted with permission.

within a formula by typing the address or moving the cursor into a cell while editing the formula, and hitting the return key would end the formula. VisiCalc also supported copying one or more cells in a range to another location using relative references. The command structure of VisiCalc was highly moded (as also shown in Figure 27.2), and commands were started with a slash "/," as Bricklin explained:

> Because the slash is an infix operator, so ... you can tell that it's there. I didn't want to use Ctrl-something, because that's something—that was commonly used in a lot of other programs of the day, that came out later, like WordStar and stuff. But that required holding something like a Shift key, and I didn't like doing that—I found that it was awkward, and it's a little harder for me because I was born with bent little fingers [shows camera], that my mom has and my grandmother had. And it sort of gets diluted as each generation comes on. So a slash got you into commands.

One command was "W" for window, which created what is called a "pane" in Section 11.2, with two parts of the data shown side-by-side that would optionally scroll together (Figure 27.3). Columns and rows of a window could be frozen so they would not scroll, which was useful if they contained titles (called "Freeze Panes" today). These features allowed working with complex large spreadsheets on the Apple II's small 24 × 80-character screen. Many future products such as SuperCalc, MultiPlan from Microsoft, Lotus 1-2-3, and Excel copied some of VisiCalc's features and UIs, sometimes making usability and functionality improvements and sometimes with steps backwards.

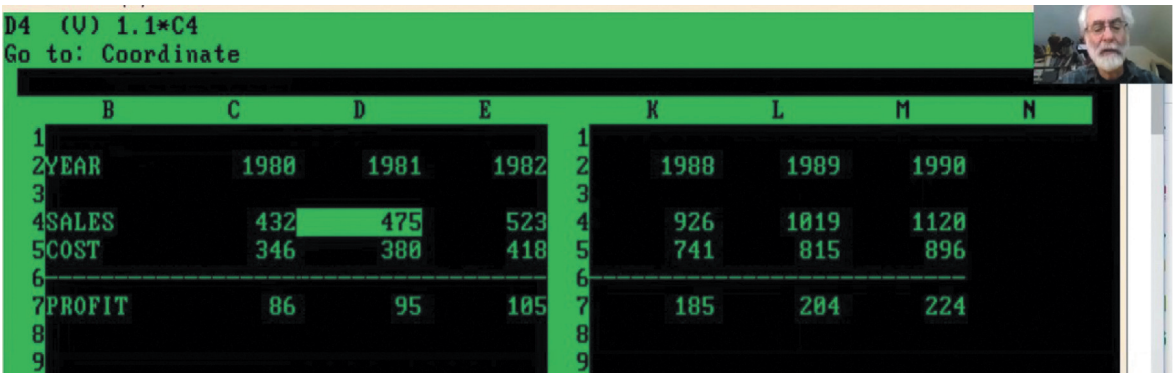


Figure 27.3 Still from Dan Bricklin's demo of VisiCalc running on an emulator of VisiCalc 1981, showing a formula for D4 (at the top) and two panes of cells. Source: Dan Bricklin, reprinted with permission.

Ben Rosen, a lead analyst at Morgan Stanley at the time, and later a funder of Lotus Development and chairman of Compaq, said in 1979 that "VisiCalc could some day become the software tail that wags (and sells) the personal computer dog" [Rosen 1979], which Bricklin said means that Rosen "as a business analyst, found it worth buying an Apple II, this weird thing." And Steve Jobs said that the spreadsheet, which originally only ran on an Apple II, was key to the success of Apple in the early days, since it opened up personal computers to business analysts (which is an interesting contrast to later when the IBM PC was for business and the Apple Macintosh was for artists and writers; see Section 2.4.7).

27.3 Dan Bricklin's Demo

One of the first prototyping tools was Dan Bricklin's Demo (Figure 27.4), first released in 1985, which was voted the best programming tool of 1986 by the Software Publishers Association. It was a DOS program that ran on a character terminal (e.g., Figure 2.7). The user could move the text cursor anywhere using the arrow keys and then just type. You could also make lines using the PC's line characters and it was smart about intersections; you could assign page changes on keyboard keys (to create what are now called *click-through-prototypes*); and you could simulate a real program, including sounds, artificial delays, and even spinning up the disk to make the noises to pretend that it was being accessed. It also had a way to overlay one page onto another so that small changes could be simulated by page

Figure 27.4 Still from Dan Bricklin's demo of Dan Bricklin's Demo II running on an emulator. Source: Dan Bricklin, reprinted with permission.

changes (before ubiquitous copy-paste was generally available). With Demo II, it could invoke programs that were written in the Demo II language, so it became what we now call an *Interface Builder* instead of just being a prototyping tool. Many significant UIs of the time were designed with Demo. Eventually the product was purchased by another company, but Bricklin was commissioned to build a version for Windows called Demo-It!

27.4 Chapter Conclusions

Bricklin emphasized that in the early days, in the mid-1970s, when he was working on word processors and VisiCalc, they had to figure out how all the interactions worked. For example, if you inserted text, should it take the style of the neighboring text or a mode based on the user? He was working on screen-based editing programs to be used by professional typists who were paid by the character and were highly trained, so it was important to minimize the keystrokes—and this philosophy was also applied to the original VisiCalc commands—but they were also designed for regular people and had to be approachable. The tradeoffs among making it powerful, efficient to use, and easy to learn persist in all UX design.

CHAPTER 28

Interview: Bill Atkinson

(a) (b)

Figure 28.1 (a) Steve Jobs and Bill Atkinson (right), taken by Norman Seeff in January 1984. (b) Screen capture from 2019. Sources: Bill Atkinson, reprinted with permission.

Guest lecture from February 4, 2019.

28.1 Short Bio of Bill Atkinson

Bill Atkinson (Figure 28.1) was the principal designer and developer of the graphical user interface (GUI) of the Apple Lisa and Apple Macintosh. He was the creator of the groundbreaking MacPaint application and its novel interaction techniques, and he also designed and implemented QuickDraw, the fundamental toolbox that the Lisa and Macintosh used for graphics. He also conceived, designed, and implemented HyperCard, the first popular hypermedia system. He invented many now-common interaction techniques, including the menu bar, the marching ants

selection used in painting programs, the selection lasso, FatBits, pull-off menus (where a menu becomes a palette), and many others.

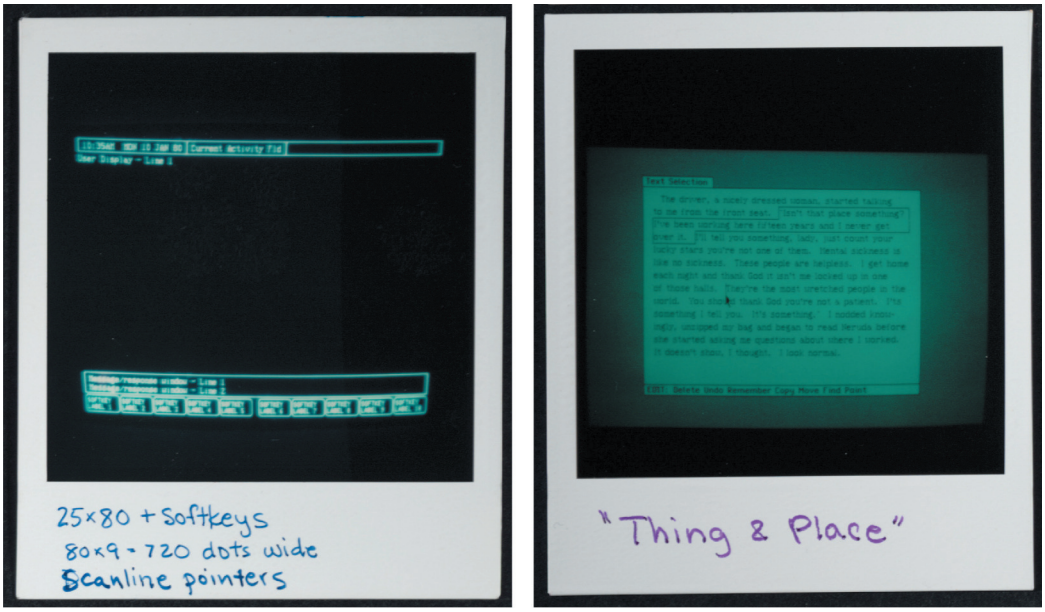
28.2 Initial Designs of the Lisa

As described in Section 2.4.7, the Lisa was the predecessor to the Macintosh, and Bill Atkinson was one of the key people working on the user interface designs and interaction techniques for the Lisa. He did much of his programming at home and took Polaroid pictures of the screens, which he would then bring in to show to the rest of the group. Fortunately, he saved these pictures and was able to show us a timeline of the evolution of the UI designs for the system.

His implementations that created the images were programmed in Pascal and assembly language (since prototyping systems hadn't been invented yet).

> Without optimized assembly language, we couldn't have afforded to do a graphical user interface, it would be hundreds of times too slow. So that had to be done in low level code, and so I wrote the QuickDraw graphics primitives that drew proportional spaced characters, drew lines, drew areas filled with patterns, because our display only had one bit display, just black or white for each pixel, so we kind of made up for that by using texture patterns. The vision was that most of the code was written in Pascal, and then lower-level things and just the graphics primitives were written in assembly level.
>
> [This included supporting] drawing into a nonrectangular area.... So I figured out how to do that, and uh, it was, well, a hundred times faster than what the SIGGRAPH core graphics things could do with polygon clipping. I worked out a new scheme called "regions".... The window manager has to kind of work in tandem with the graphics to do the right clipping to ... keep it within the visible region of that window [even if it was covered].

For *menus*, based on the Apple II, the original designs used softkeys (Figure 28.2(a)), but these were quickly abandoned as too error prone—since applications needed more than 10 operations, the soft key operations had to frequently change. Also like the Apple II, they originally tried a white-on-black screen (Figure 28.2(a)), but switched to black on a white background (Figure 28.2(b)) because they wanted to support photos and graphics which don't look right when reversed. After seeing the Smalltalk system (see also Section 2.4.3.1) and other demos from PARC, they investigated popup menus, but they abandoned that and tried a menu along the bottom (Figure 28.2(b)) where clicking on the "Edit" on the left would popup a menu of the other top-level options.



(a) (b)

Figure 28.2 (a) Initial ideas for softkeys and white on black. (b) Black on white text, with "Thing & Place" interaction, and initial menu bar at the bottom. Sources: Bill Atkinson Polaroids, reprinted with permission.

The next design was more like today's menu bars, with the top-level options all displayed at the top and the submenus appearing below them, but the menu was in the window. However, when the window was too narrow, you couldn't see all the items, and the dropdowns didn't display when the window was at the bottom of the screen, so Atkinson said he tried moving the menu to the top of the screen, but "everybody bitched. They complained, oh this is terrible, it's too far away, it'll be hard to reach," so they added a gain function where moving faster increased the distance moved by the cursor (Section 4.6). They also pinned the cursor at the top of the screen, instead of allowing it to wrap around to the bottom when it goes off the top. Another advantage of having the menus at the top was that:

> ...there's sort of a kinesthetic memory of where something is. "Oh, the file menu is over here." ... So, even before you really remember what command you're looking for, you start the movement in that direction already. Your kinesthetic memory was leveraged there. And that could really only be done by having the menus at the top of the whole screen, so ... they were in a fixed location.

This is one of his biggest joys of the design:

> I was pretty excited when I moved the menus to the top of the screen, and then it just clicked. Like okay, we have the full width, ... this kinesthetic reaching over for this place. ... I thought, yeah! This is gonna work.

One abandoned idea was to have a selection (called a "thing") and a separate cursor (called a "place") so you could copy the thing to the place (Figure 28.2(b)), but they did user testing on their spouses, who were confused by being at two locations at the document at the same time, so they went to the cut-paste model from Larry Tesler.

For *text editing*, they experimented with having the cursor be the selected character, like on the Apple II, but eventually adopted the cursor in between characters. To help people understand that, they changed the pointer cursor to be a vertical bar (I-beam) rather than a pointer.

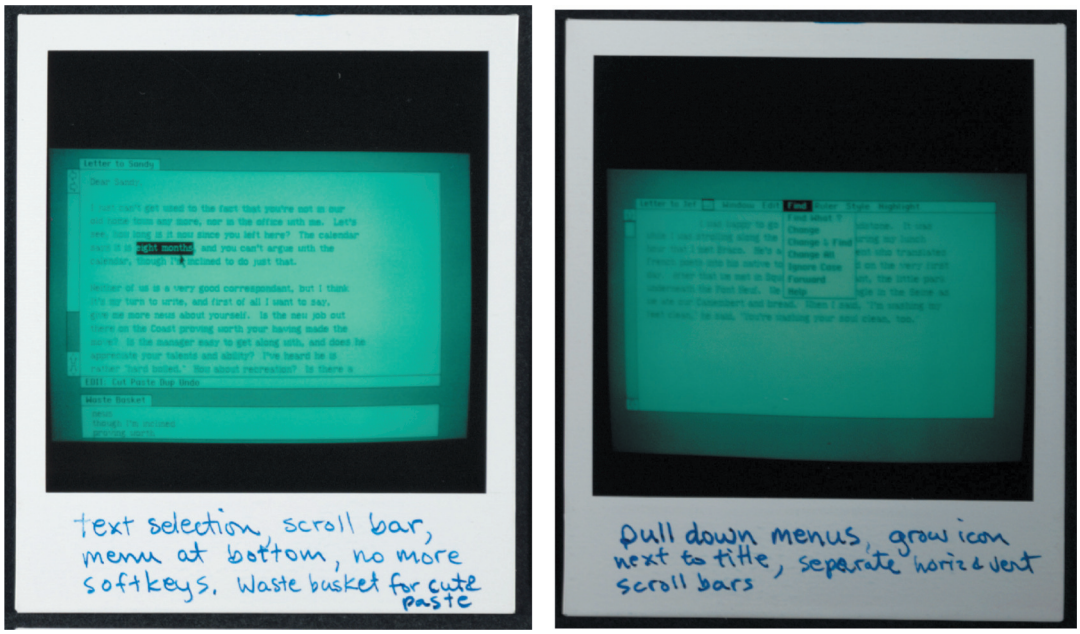
For *windows*, the titles originally looked like tabs (Figure 28.2(b)), like in Smalltalk, but they went to full-width when the menu bars moved to the top (see Figure 28.3(b)). Originally the resize icon was in the menu bar (see Figure 28.3(b)), but users accidentally made the window tiny, which was confusing. Therefore, they moved the grow icon to the bottom right (Figure 28.4(a)), so a small movement or slip would only change the size a little. This also meant that there always needed to be both a horizontal and vertical scrollbar, and they should be on the right and bottom (instead of on the left as in Smalltalk, Figure 28.3(a)).

For *scrolling*, they experimented with a proportional thumb (see Figure 28.3(a)), but when most of the document was showing, then the thumb would fill most of the scrollbar area, and:

> "Our experiments with users were that they didn't get it, that that was an affordance that you could touch on. It now was just this empty space. They would click over here [in the non-thumb grey area of the scrollbar], but they wouldn't have any idea that this [the thumb] was something you could click on. We also added arrows for moving a little up or a little down, or a little left or a little right.... [So] we went back to a non-proportional affordance, this [thumb] is the thing you touch on and drag if you want to scroll, and it's no longer proportional to the portion of text that's visible." (see Figure 27.3(b))

They tested whether the scrollbar arrows should point in or out, but users were split 50-50.

For the *mouse*, they convinced the managers of the Lisa project to require having a pointing device, so the software could always assume it would be there.

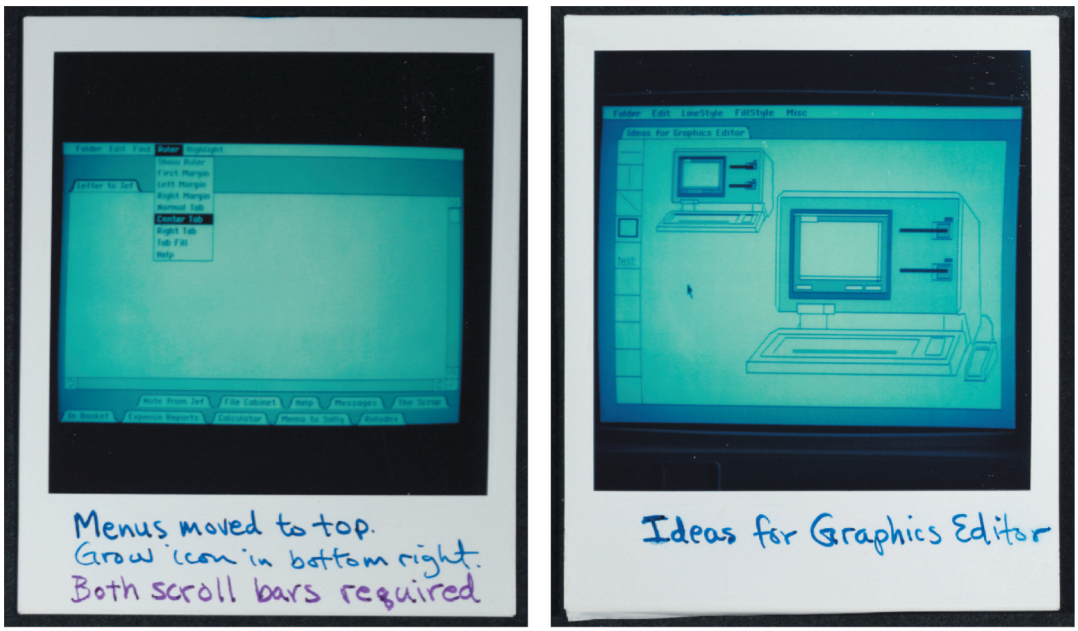


(a) (b)

Figure 28.3 (a) Early scrollbar experiment on the left and the "Waste Basket" visible at the bottom. (b) Initial menu bar design, with the resize icon in the menu bar. Sources: Bill Atkinson Polaroids, reprinted with permission.

> We didn't want to have to write a graphics editor ... that might have to work only with cursor keys. So we did commit to having a mouse, after going through we checked out tablets and we checked out touch. Touch systems really didn't work at all at that time. They didn't really have anything like what we have on an iPad right now, it's a very gentle touch. What they had back then, you had to really press and you couldn't drag, and it was not really usable. So we went with a mouse. We had longer discussion about well, how many buttons should be on the mouse? Some of the mice we had seen from Xerox PARC had three buttons.... And then we tried sort of user testing with this, what we found is as long as you had more than one button, people spent a lot of time—this is beginners— ... looking at the mouse before they would touch it. When it was only one button, they stopped looking at the mouse entirely and were just looking at the cursor on the screen. And so that's the main reason we went to one button, is to simplify it there.

Atkinson related a fun story about *user testing* and the *origin of the "OK" and "Cancel" buttons*:



(a) (b)

Figure 28.4 (a) Updated menu bar, scrollbars to the right and bottom, grow icon at the bottom right. (b) Early design for tool palette that supports drawing modes. Sources: Bill Atkinson Polaroids, reprinted with permission.

> [A] little bit later when we had actual text editing and things like that, ... we would put up a dialog that would say, "Do it" or "Cancel", and we'd give somebody a document and say, "Here, edit this and save it," and they'd get to the point where they're supposed to choose "Do it", and they'd look a little miffed, and then hit cancel. And we saw that several times, different people. And so we had in the ceiling a video camera on the keyboard and mouse, and another on the screen with their face reflected in it, and we actually synced those two up, and we asked them to mutter under their breath, stream of consciousness, what are they thinking about, what are they doing. 'Cause actually it's more interesting to know why somebody did something than just what they did. And when we saw this, people kinda looking a little miffed and then hitting cancel instead of do it, we turned up the volume and played it back, and: "What's this 'Dolt'? I'm no dolt! So I hit cancel." So, we knew that it said D-O-uppercase-I-T. They read it as D-O-lowercase-L-T. We couldn't possibly read it as dolt, because we already knew what it meant—it was "do it!" To them, they read it as dolt! So, what we had to do was change it, to OK and Cancel. Now, we didn't like OK, because it's an abbreviation

> and a colloquialism and, and, but we did.... Well years later, I would say it would have been much better to write "Save" and "Cancel"—right? ... And the reason is that when you have a dialog with some text there, and then a yes or a no kind of thing, they don't read the text! It's too small, they're in a hurry, they just wanna go for the buttons. So the buttons have to have the essence of the meaning in them. That's why "Save" and "Cancel" would have been much better than "OK" and "Cancel." But we didn't learn that at first. So that's where some of these ideas come from, is just brute force, trying different things and see what works.

On *user testing*, he further explained:

> We used lots of testers. We'd bring in one subject and run a bunch of tests on them, and then they were, they were done, and we'd bring in somebody else. You can find out most of what you're going to find out in the first five testers. You don't need fifty testers to find out something like "Dolt". ... But what you have to have is the humility to realize that it's not their problem, it's yours. You wrote the software, they don't understand it. It's not that they're stupid, it's that you have misled them. And you have to figure out a way to not mislead them. So yeah, we used a lot of testers, but ... each one we'd run different things through, and then we'd let them go. We tried to have mostly beginning users, that were approaching this without a lot of previous experience.

Atkinson explained that the *tool palette* was a reaction to taking Tesler's "no modes" philosophy too far (Figure 28.4(b)):

> When Rod Perkins was working on the Lisa graphics editor, he would stretch out a rectangle of selection, a marching ants rectangle, and then say, "put an oval in it." But you wouldn't see the oval in it while you were stretching it out. Or you would say, stretch out a rectangle and put a line in it, or put a rectangle in it. So, I came in one day and said well, you know, this is just brain-dead. If you're drawing something, you first pick up a crayon, or you pick up a Sharpie, or you pick up a pencil, or you pick up an eraser. And then you do a bunch of stuff with that eraser. And so, you'd need to be in a mode. I thought the creation tool in graphics editing needed to be in a mode, but we wanted that mode to be in front of your face so it wouldn't be confusing. And so that's where the first tool palette came from. So ... you could dip into horizontal lines, and now anything you did with the mouse would make a horizontal line, or dip into diagonal lines and you could make diagonal lines, or dip into rectangle, or dip in and you could make text. And I brought this in to the Lisa team when they sat down [and] ... just try some stuff. Here, use

> this, dip and draw, dip and drag. While you were dragging out a rectangle, you were seeing the rectangle. While you were dragging out a line, you were seeing the line. It did require that you be in a mode, but that mode was made visible by this tool palette that highlighted the one you're in. And the reaction, you know, was a lot of forehead slapping, duh, of course, right? And ... maybe 20 people were gathered around looking at this and—yeah, of course. So we realized that our dogma about no modes was causing us to be blind to what you know. We weren't thinking, we're letting the dogma of no modes cause us not to design in a straightforward fashion.

Drag and drop was brought into the Lisa as a "skunk works" side project to come up with a better way to manage files and folders on the desktop (the Lisa "Filer" or Macintosh "Finder"). This also introduced the trash can icon on the desktop, and "all of this work happened in less than two weeks." The design required Atkinson to add icons as a primitive graphic type since it needed both the picture to display and a mask to allow parts to be see-through. Since the background was a half-tone pattern, icons would look different on even or odd pixels, so they moved them onto a grid so they would look consistent. At the time, the UI was supposed to be locked, but this change was snuck in less than 6 months before Lisa shipped.

Another story was about how the *accelerator keys for menus* came about (see also Section 15.3):

> Initially we were gonna have, on the keyboard, cut, paste, copy, paste, undo, and redo keys. Hard keys. And we ended up backing off from that.... At first, all the commands were available only from the pull-down menus. And I went to visit Doug Engelbart, and he showed me his Augment system, now this was a system which was designed for professional knowledge workers, and it didn't matter that it took years to learn how to use it, it gave them power. And he said, as you're working on this user-friendly system, be careful in your desire to make it usable by somebody just walking up to it, be careful to not lock out the power user. When someone gets proficient at using this thing, you don't want to slow them down. As an example, he said, look how hard it is to ride a bicycle, but it's worth it. Bicycles are not beginner-friendly. You crash a lot. And then eventually you learn to keep it balanced, but it's worth it because it really amplifies your leg muscles, and you can go much further than you could go walking. And so with that in mind, I went home and that night, I added the command keys onto the pull-down menus. So, when you pull-down a menu, over on the right there were letters that you could use, we [used] a key on the keyboard—the command key—if you hold command down and then type that letter, then it would be a shortcut for doing that

> menu. How we specifically ended up with ZXCV [for undo, cut, copy, paste], was so we wanted it to be on the left hand, because your right hand for—most people typically are right-handed—typically your right hand is on the mouse, ... and then the keys that were easy to reach are those that are right adjacent to it, ZXCV. And C was sort of a natural for copy. V sounded like an insertion—you know, it looked like an editor's mark for "insert here," so that was paste. And X, looked like an "X it out," that's cut, and command-Z was, we used that for undo and it didn't have a mnemonic, it was just those four characters were the easiest ones to reach.

Atkinson explained how the *Command icon* on the keyboard came about:

> It actually was an Apple logo [on the Lisa], and later when we did the Mac, Steve Jobs hated abusing the Apple logo by using it for that function. So, he asked Susan Kare to come up with an icon for command or feature, and that little clover-leaf of freeway interchange symbol that she chose came from a Finnish map, and it was the symbol for an outstanding feature, like oh maybe there's a statue here, or a park here, or you know, something like that. So she sort of used "Feature."

28.3 HyperCard

Atkinson read a long passage about how HyperCard (see also Section 2.4.9) was inspired by an LSD trip in 1985.[1] With respect to the HyperTalk language, he said:

> The design goal on the HyperTalk was to make it readable. Not necessarily writable, but readable to a nonprogrammer.... You got a HyperCard stack, you could open up a button and see, it says `onmouseup goto nextcard`. It was designed so that somebody who wants to be a programmer could look at this and see how they could modify it for their own needs—they could tweak it and write their own. ... I think of it as event-based. ... Programmers think, if–then–else. But non-programmers think action–reaction. You poke it here, it does this. You know, you pet the cat, it purrs. You push the cat, it scratches you. So, instead of having if–then–else things, we just have buttons that do something when you poke them.

1. The passage he read was from an April 2016 interview by Leo Laporte that is available from: https://twit.tv/shows/triangulation/episodes/247. The part about the inspiration for HyperCard is in my transcript of Atkinson's talk, and also here: https://www.mondo2000.com/2018/06/18/the-inspiration-for-hypercard/.

> You could write software without having to be a programmer, without having to have a compiler, linker, code generator or all that stuff, and had high-level structures that saved you a lot of work. Like, when you just dragged and dropped a field there, you could type in the field and close the stack and it would stay there, it would be there, so all of that persistent memory stuff was handled for you, so you weren't dealing with [saving] your state out to a disk or something. ... One of the criteria I had was, you should be able to unplug the computer, and lose no more than a few seconds of work. And so what I did for that, was to always trickle a journal out of what's happening.

Atkinson said: "Chris Espinoza named it HyperCard, because of the feature of linking from one card to another, like Ted Nelson's Hypertext." In summary:

> I think HyperCard was successful in enabling passionate people to express themselves, and I always thought it was easier to give programming ability to someone with the passion, than to take a programmer and try to instill passion. I would say that, of the 12 years I spent at Apple Computer, what I did was make tools to empower creative people. That's really sort of a summary, what I was trying to do was empower creative people to express themselves—whether it's MacPaint or HyperCard.

28.4 General Magic

Of his next venture after Apple, Atkinson described what he did next (shown above in Section 3.9):

> I left Apple after 12 years, and I co-founded a company called General Magic, we wanted to make personal communicators that would be intimate devices, with you all the time. Apple was making computers and they're not interested in making consumer electronic devices. They wished us well and spun us out, but they weren't interested in what we were doing. We wanted to do something smaller that would be in your pocket and with you all the time, and we failed. We really couldn't—the components were too expensive, there wasn't a good capacitive touch sensing that you could do gentle swipe. And the meteoric rise of the Internet drew the attention away from it and all our partners kinda went to work on the Internet, and they were right to do so. We were just ahead of our time. I'm pleased to see some of the ideas that we had bore fruit in the long run, and that's really what I saw, and the first iPhone was hey—somebody's finally done the personal communicator.

CHAPTER 29

Interview: Bruce Horn

(a)

(b)

Figure 29.1 (a) From left: Randy Wigginton, Jerome Coonen, Rony Sebok, Donn Denman, Andy Hertzfeld, Bruce Horn, Bill Atkinson, Susan Kare, Owen Densmore, Steve Capps, Larry Kenyon, Patti Kenyon, Tracy Kenyon, Steve Jobs, taken by Norman Seeff in January 1984. (b) Bruce Horn more recently. Sources: From Bruce Horn, reprinted with permission.

Guest lecture from February 17, 2016.

29.1 Short Bio of Bruce Horn

Bruce Horn (Figure 29.1) was a member of the original Apple Macintosh design team at Apple Computer, where he created the Macintosh Finder and the Macintosh Resource Manager. His signature is amongst those molded to the case of the Macintosh 128K, and his name was mentioned in the 2015 Steve Jobs movie. Bruce received a B.S. in Mathematical Sciences from Stanford University in 1982 and an M.S. and Ph.D. from Carnegie Mellon University in Computer Science in 1994. He was a student in the Learning Research Group of Xerox PARC where Smalltalk was developed starting while he was still in high school (1973–1981), and he overlapped at PARC in the summer of 1976 with me when I was a summer intern there from MIT. While there, Bruce worked on various projects including the NoteTaker, a portable Smalltalk machine, and wrote the initial Dorado Smalltalk microcode for

Smalltalk. Later, Bruce was employed by Powerset as the principal development manager of the Natural Language Technology group. Powerset was acquired by Microsoft in the fall of 2008 and is part of Live Search. From 2012 until 2018 Horn was an Intel Fellow and Chief Scientist for Smart Device Innovation in the New Devices Group at Intel Corporation. As of June 2022, Horn went back to Apple as a Distinguished Engineer, Siri and Information Intelligence.

29.2 PARC and Smalltalk

An important early influence on Bruce Horn was his work at Xerox PARC in the Smalltalk team. He grew up in Silicon Valley and got to start working there when he was only 13 years old. He wrote to Hewlett Packard and applied for a job when he was 10, and they said HP didn't hire people under 18, but Xerox did. Ted Kaehler from Xerox came to Horn's middle school to give a talk, and through him, Horn got to be in the Smalltalk group with Kaehler, led by Alan Kay. Alan had the idea for a *Dynabook*, which was not for consuming "passive" media like the iPad but rather for what Kay called *Personal Dynamic Media*. The idea was to enable powerful ideas by allowing people to construct simulations of the world. This was a super radical view of the world at the time—having a super lightweight computer for children was out of the blue and impossible with the technology of the time (1970s). As presented in Section 2.4.3.1, Smalltalk innovated windows, menus, using the mouse, modeless editing, and many other interaction techniques, which Horn helped with.

Adele Goldberg and Dan Ingalls in the Smalltalk team were ordered to show Smalltalk to the Apple visitors, including Steve Jobs and Bill Atkinson. During the demo, Steve Jobs said he didn't like the way the text scrolled, so Ingalls said OK, and went into the code browser and made a couple of changes and all of a sudden there was smooth scrolling text, which no one had ever seen before. It blew them away. Horn quoted Larry Tesler saying:

> Jobs was pacing around the room ... he was very excited. Then, when he began seeing the things I could do onscreen, he watched for about a minute and started jumping around the room, shouting, "Why aren't you doing anything with this? This is the greatest thing! This is revolutionary!"

Horn further quotes Jobs saying:

> They [Xerox] just grabbed defeat from the greatest victory in the computer industry... If Xerox had known what it had and had taken advantage of its real opportunities, it could have been as big as IBM plus Microsoft plus Xerox combined—and the largest high-tech company in the world.

At Xerox, Horn and others were frustrated, believing management would never make anything of the technology that they created at PARC, but they saw that Steve really got it, and that Apple was the place where they could actually deliver this to the world. So, a handful of them ended up going to Apple in 1981, and as one writer said, "stealing the fire of the gods and giving it to the people." Horn was very proud that they did that.

29.3 Apple Macintosh

Horn worked in the original Macintosh team which at the time was fewer than ten people (see Figure 29.1(a)). They were all young and mostly unmarried, so they were able to work around the clock on it. Horn said:

> Steve was passionate about shipping this technology. We had sweatshirts made that said, "90 hours a week and loving it!" We erected a pirate flag on our building because Steve considered us pirates, stealing all the good ideas for the Mac. ... And we were like family. The Mac team worked seven days a week and had lunch and dinner together every day. It was a true mission and we were committed; we had all been influenced by Steve's famous "Reality Distortion Field" where everything he said we could do, no matter how outrageous, somehow seemed possible.

In contrast to what Bill Atkinson said about the Lisa development (Section 28.2), Horn said that they didn't do any user testing on the Macintosh—they just built something that they liked. There were lots of little design decisions and innovations in the Apple Lisa and Macintosh that Horn mentioned, most of which are still being used today, more than 30 years later.

- In the original design, when the system opened, the main disk would be highlighted and if the user just started typing, it would rename it. So Horn said he had to fix that.
- Ellipses were added (e.g., to menu items) to indicate that the user was not done, that "I need more information."
- Horn created a system called the "Resource Manager" that could configure all the assignments, so they could trivially change the code to be localized to a different language.
- They wanted to make sure that people could explore. Undo was important so the user could get out of trouble. They supported as much being undoable as possible, so the user could try anything fearlessly.

- Disabling menu items was important to make clear that the user couldn't do that operation. Like a guard rail, it showed that you can't do this now. Popping up an error dialog was inelegant and seemed Windows-like, so they didn't do that.
- Disabling menu items led to Balloon Help to let you know why it was disabled. But it was too difficult for software developers to provide, so they just stopped caring, and it disappeared.
- Smalltalk would redraw the whole window when it came to the top. Bill Atkinson didn't realize it wasn't doing the right thing of redrawing only the exposed parts, so he had QuickDraw use "regions" that could allow drawing only in those exposed parts, which could be a collection of rectangles. That made the Apple windows work beautifully.
- PC Manuals were like 3ft tall, but Mac was very slim in contrast.
- Susan Kare did all the graphics and fonts. They created a new font, called Chicago, which is still used today. It was very bold and blocky since it was designed to be easily read even when it was inverted or greyed out using a halftone pattern. "She did a tremendous job."
- Drag and Drop filing and copying—if drag-and-dropped on the same disk, then it would move, but if drag-and-dropped to a different disk, then it would copy. They wanted to make people have to deliberately delete it from the original disk.
- Horn invented the concept of "preflighting" for drag and drop to determine whether the file copy would succeed based on whether there was enough space in the destination. That way, users would never end up with a partial copy of the files that were selected.
- Applications would come back to the way they were the next time you opened them, even if you had closed them. Everything was saved in a database.
- They wanted to make it easy to name documents, including with spaces, with no file extensions. So, they had the Type/Creator property associated with each document as a four-character code. Applications published the kinds of documents that they knew how to open, so documents knew who owned them, but users could drag and drop document icons onto icons for applications, even if it wasn't the owner of that type, as long as the application indicated it could open that kind of document.
- The trashcan also contributed to making people comfortable. You want to throw stuff away, but not right away. This design made you do an extra

step—you had to put something into the trash, and then empty the trash to get it to actually be removed.

In a later email, Horn summarized:

Radio buttons (definitely) and check boxes (I believe) were in the Smalltalk system at PARC. Ted Kaehler coined the term radio buttons after the buttons on old car radios, as you probably remember:-) I don't know about the invention of the menu bar—that was from the Lisa team. We (on the Mac team) invented a few of the menu interactions like ellipses to indicate a following dialog box, flashing to indicate selection, hierarchical menus and so on.

CHAPTER 30

Interview: Rob Haitani

Figure 30.1 Rob Haitani. Source: From Rob Haitani, reprinted with permission.

Guest lecture from April 10, 2019.

30.1 Short Bio of Rob Haitani

Rob Haitani (Figure 30.1) was a Product Manager and UI designer on the original Palm Pilot. He is the co-author of the definitive article about the UI design of the Palm in *Information Appliances and Beyond*, E. Bergman (Ed.) [Bergman and Haitani 2000] and in the section on portable devices in Bill Moggridge, *Designing Interactions* [Moggridge 2007]. Currently, he is a Senior UX Manager on the Alexa team at Amazon in Japan.

30.2 Casio Zoomer

The Palm company was originally a pure software company, and it was creating all the applications for a product called Zoomer from Casio when Rob Haitani joined the company. It felt too slow, which originally was blamed on the processor speed, but he compared how to perform common tasks on the Zoomer with the Apple Newton (Figure 30.2), and they all found common tasks, like creating a meeting,

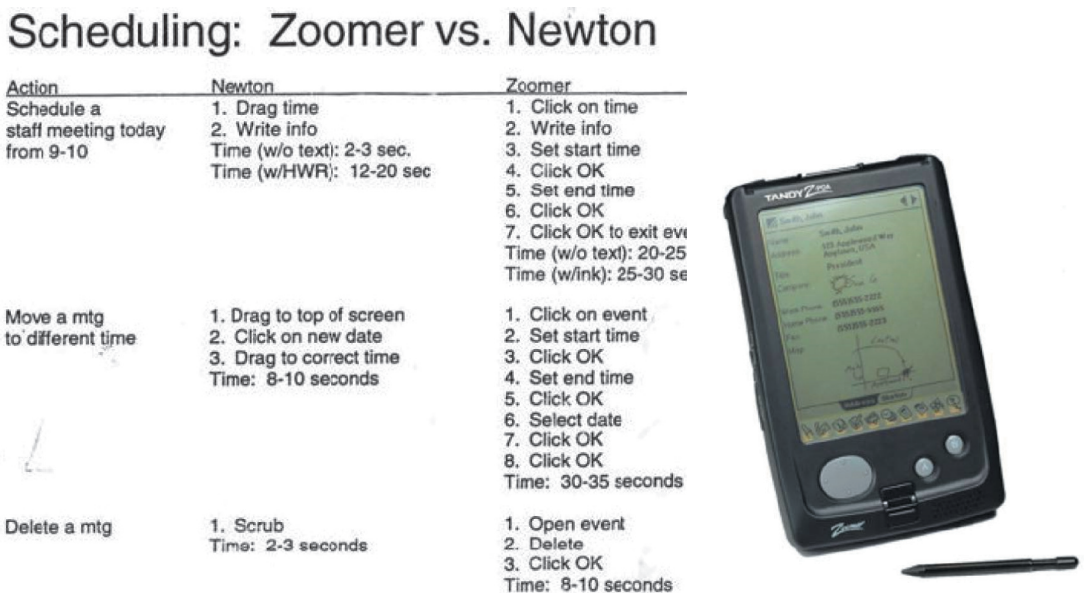


Figure 30.2 Haitani's comparison of the Zoomer device vs. the Apple Newton (pre-Palm Pilot). Source: From Rob Haitani, reprinted with permission.

took way more steps, which inspired Haitani to focus on being a user interface designer since that can have so much impact.

They also developed the Graffiti gestural language (Figure 3.18(b)) for the Zoomer. A key observation was that "absolute speed and accuracy aren't as important as overall experience," so, for example, they made the backspace gesture be a simple horizontal stroke to the left, which was very fast. In contrast, on the Newton, if it was wrong for handwriting recognition, you had to scribble out the text or laboriously select it with a lasso tool. Another advantage was that "with Graffiti you could write in a heads-up manner. ... It was so forgiving, you could hold it in your hand and write and be looking at the person you're talking to, ... unlike today, where we're all heads down in our phones."

The Zoomer and the Newton were both unsuccessful products. They were also both large and heavy. The second version of the Newton, called the MessagePad 2000, was even larger and heavier than the first.

30.3 Palm Pilot

Jeff Hawkins, the founder of Palm (see also Section 3.13), felt that they were going the wrong way—getting faster, bigger, and heavier. So, they took the opposite approach and designed a device with a simple OS so it could use a slow, low-power

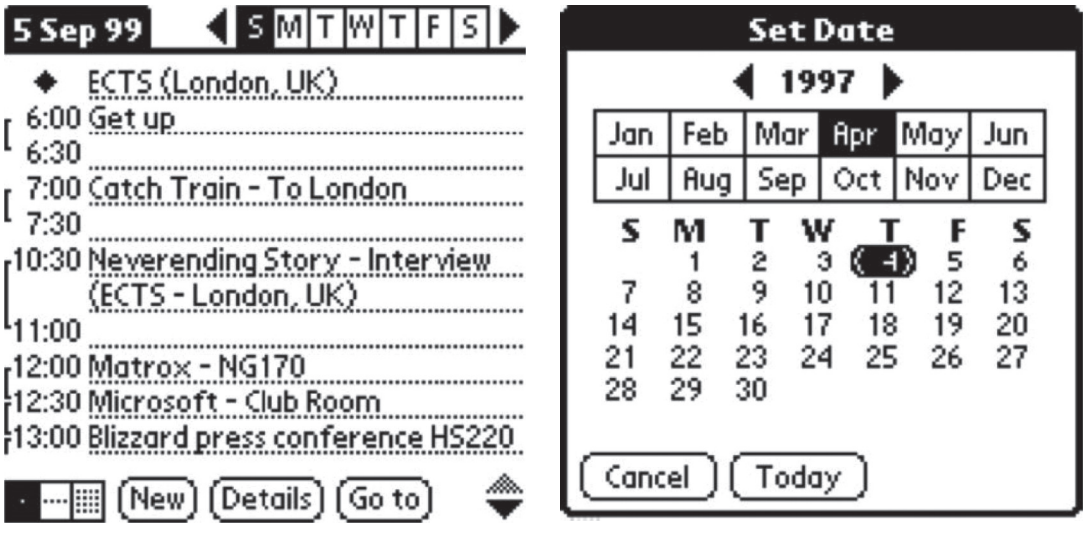
processor that was too slow for animations or drag-and-drop. They decided to design the UI and the apps first and decide on the minimum OS they could get away with later. The design was all created using HyperCard and MacPaint so they could prototype and explore the designs before the hardware was even ready. Haitani said:

> [The design relied] really heavy on the prototyping. ...We actually did a full test of the interactive HyperCard prototype before any of the code was [written]. ... You want to use usability testing with functional devices, but by the time something's coding and working well enough, then it's kind of too late to change things, so we did rely heavily on the prototype.

The screen was monochrome 160 × 160, which is smaller than the number of pixels in a single app icon on the current iPhone. At 160 × 160 every pixel matters. One strategy was that they saved space by having indicators also be controls, like the day of the week at the top of the calendar screen (Figure 30.3(a)). Haitani designed the fonts pixel-by-pixel to use the minimal width—he couldn't get it to be smaller than 5 pixels, but that was 23% more text than using 6 pixels per character. The numbers were all the same width so they would line up nicely.

The main principles were "*Fast, Simple, Intuitive*."

By "*Fast*," there was not allowed to be a wait cursor and nothing could cause the system to hang up. "Fast" also applied to the app designs—for example, in



(a) (b)

Figure 30.3 Original Palm calendar screens. Sources: From Rob Haitani, reprinted with permission.

the calendar day picker (Figure 30.3(b)), any day in the year can be reached with just two taps. There is also no "OK" button—this dialog just goes to the day when the user taps on it, and you can recover with just one tap. In general, they tried to maximize the number of things on each screen while at the same time keeping the screens simple. A key observation that made this possible is that all *frequently used* actions needed to be fast and take as few steps as possible, but only *a very few features are used frequently*. For example, for contacts, you often need to look up a person by name, create a new contact, and view details, so these operations were all available from the front screen, whereas editing and deleting are less frequently used, so these are in sub-dialogs even though this seems like a consistency problem. Haitani said they mostly identified which were frequently used through internal design meetings and their own intuitions. They did some user tests using the HyperCard prototypes when there were disagreements in the team, such as avoiding modes for editing the text in-place.

In terms of "*Simple*," Jeff Hawkins thought of the idea of having the digitizer continue below the screen since it was too expensive and too much power to have a larger screen. On the Zoomer, they had a whole row of icons at the bottom (Figure 30.2 at right), but with the Palm they eventually decided to just use four and put them on physical buttons (Figure 30.4(b) and (c)). Similarly, they eliminated the need for sideways scrolling so they only had up and down buttons. They even eliminated the scrollbar in the first version to save 6 pixels at the edge, and they just used up and down arrows on the screen (Figure 30.4(c)). Originally, they had modes for numbers, with buttons to switch (Figure 30.4(b)), but realized they

(a) (b) (c)

Figure 30.4 Iterations of the screen designs. Sources: From Rob Haitani, reprinted with permission.

could save a tap and avoid mode errors by just having two fixed areas, the bigger one for letters and the smaller for numbers (Figure 30.4(c)).

The design for the Palm Pilot was to be more like a watch than a laptop—pull it out frequently but to only do short tasks like on an organizer. At the time, you wouldn't try to do long tasks. This was in contrast to Microsoft:

> "[O]ur main competitor, after the Newton went away, was Microsoft and their whole philosophy was, to take the desktop computer design and shrink it into ... a handheld device. So you want to have your familiar Windows interface, ... —that was their whole value proposition, that your Windows interface was shrunk down into a handheld device. And we were taking a radically different approach [which he called]...'don't shrink; rethink.'"

Another important simplicity feature of the Palm was "HotSync." In the original Zoomer, the philosophy was to free you from the computer, so syncing was only available as an extra-price add-on, and at the time, the Apple Newton's synchronization was hard to make work at all. However, with the Palm Pilot, they realized how important synchronizing calendars and contacts with the computer were. It needed to be easy to do, but at the time they couldn't just synchronize in the background while it was connected with the wire since it was considered to be slow and unreliable, so people needed to be reassured that it happened by seeing the progress and the confirmation. Also, since there was only RAM and no Flash memory at the time, if the battery died, all the information was lost, which made synchronization even more important.

For "*Intuitive*," they adopted existing and familiar paradigms from computers like pull-down menus and checkboxes (but they didn't have a radio button for screen space and just highlighted the current choice or used a popup). They also diverged from the way other systems worked when they could make it simpler. One example was to edit strings, like in the Palm calendar, where users could just tap and start typing, and hit return or tap elsewhere to finish. This avoided needing OK–Cancel and extra taps, but it made it more difficult for the programmers to implement.

Another interesting design decision was that the physical buttons would all turn on the device if it was off, and they would always take you to the most useful screen. So, pressing on the Calendar button would turn it on and go to today's date showing the current time. Note that this is not symmetric since these buttons could not be used to turn it off—that could only be done by the dedicated power button.

In closing, a key observation was that when users say "I just want to...," that is a signal about what the key functions are that need to be especially easy and quick to do.

CHAPTER 31

Interview: Tandy Trower

Figure 31.1 Tandy Trower. Source: From Tandy Trower, reprinted with permission.

Guest lecture from March 16, 2016.

31.1 Short Bio of Tandy Trower

Tandy Trower (Figure 31.1) has been a leader in the creation, marketing, and deployment of a wide variety of consumer and business products as well as operating systems and developer tools for over 30 years. At Microsoft, Trower managed a wide variety of new initiatives and products that included the company's programming languages, the first releases of Windows, and applications for the Apple Macintosh, as well as hardware products and games and educational software products. He also established Microsoft's first usability labs and user-interface design services organization. In more recent years, he served as the founder and General Manager of the Microsoft Robotics group that created Microsoft Robotics Developer Studio. He is now the Founder and CEO of Hoaloha Robotics.

31.2 Precursors to Microsoft Windows

Tandy Trower first started working for Microsoft in 1981, and he at first worked on compilers for programming languages like Microsoft Basic and Microsoft Pascal. He also worked on early games including the Microsoft Flight Simulator. The first "windowing" interface he worked on was the Microsoft Interface Manager from 1983 (see Figure 31.2), which tried to do Microsoft DOS applications in a bitmapped way. Trower said he was not personally involved in the UI design of that product, but that:

> A lot of it was inspired by the Xerox Star. ... Bill had purchased a Xerox Star and installed it in an office all to itself, a few doors down from me, and he had sent email throughout the company encouraging people to experiment with this, that he saw this as the future of computer technology and this whole idea of bitmap screens and a mouse to interact with things on the screen were totally revolutionary.

31.3 Microsoft Windows 1.0, 2.0, and 3.0

Trower went over to work on Microsoft Windows on January 2, 1985:

> [The UI of Windows was] heavily influenced by a newly hired development manager, by the name of Scott McGregor. Scott McGregor also came from Xerox PARC. But he was developing its own version of a bitmapped interface called Cedar, and you can see here that one of the major differences between Cedar and the Xerox Star interface was that Cedar lined all the application

Figure 31.2 Microsoft Interface Manager screens from 1983. Source: From Tandy Trower, reprinted with permission. Used with permission from Microsoft.

> windows next to each, and so there were no overlapping windows except for dialog boxes.

Cedar was designed for a large screen (see Figure 31.3 and Section 2.4.3.4) and even though the IBM PC had a tiny screen (320 × 200 pixels, 16 colors, like an average TV set), Microsoft still used Cedar's tiled interface for version 1.0.

> And the mouse was considered to be an optional interface to the thing. Windows was designed with the objective initially to be able to allow the user to navigate the entire interface from the keyboard. This was a pretty hard thing to do, and it was often very crude to try to do this. In any case, I stepped into this project, which was already considered [to be] embarrassingly late, given the label "vaporware," because there were many industry pundits that said that Microsoft would never ship it.

It had been announced in 1983 for delivery in 1984, and Steve Ballmer, who was Trower's boss, said that Trower's job was to ship it by summer of 1985, even though it was only half-finished when Trower joined the team 6 months before that. They did ship "Windows 1.0 Premiere Edition" in August of 1985 that was available to the press and to OEMs (Figure 31.4). Originally, the IBM PC was shipped with a different OS, and Microsoft had to try to get consumers to buy Windows separately and install it. So they created a series of print ads that showed off the primitive graphics they could do, focusing on the use of color, which the Macintosh didn't

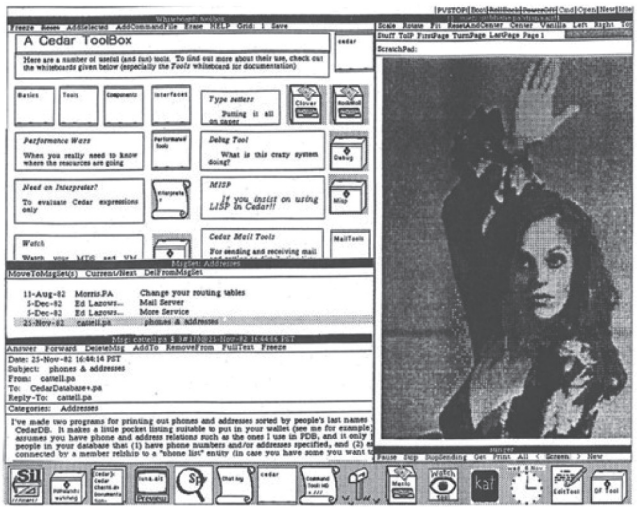


Figure 31.3 Cedar environment from Xerox PARC, which Scott McGregor worked on. Source: From Tandy Trower, reprinted with permission.

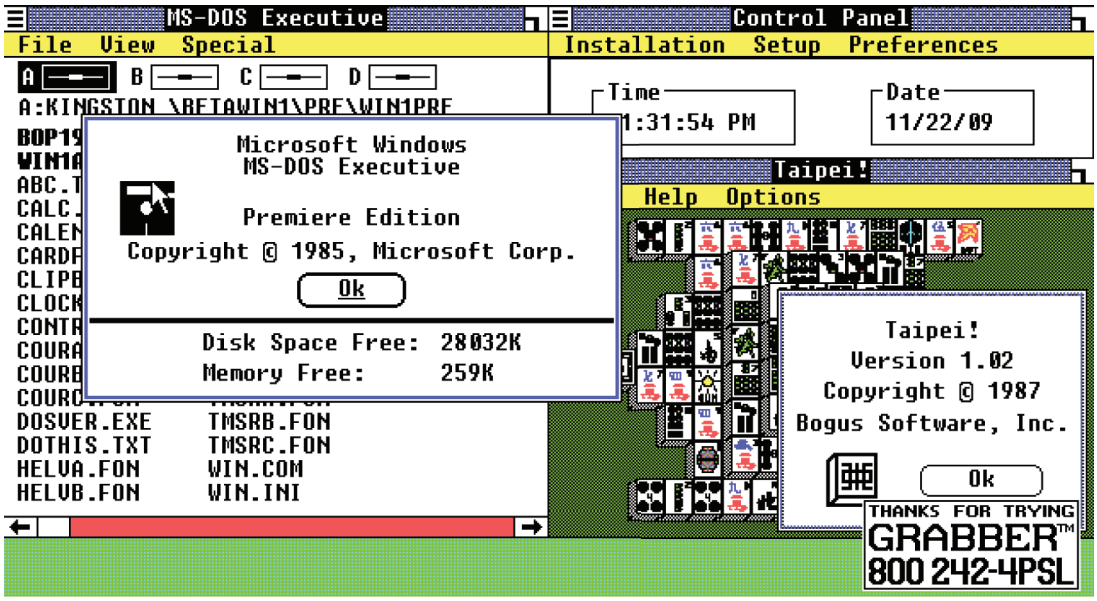


Figure 31.4 Windows 1.0 Premiere Edition released in August 1985. Source: From Tandy Trower, reprinted with permission. Used with permission from Microsoft.

support at the time. It also highlighted that you could run multiple applications at the same time, which the Mac also didn't support.

The OS that IBM was shipping at the time was called TopView and was character-based (not bitmapped), and it also supported multiple applications (Figure 31.5). The key applications at the time were the Lotus 1-2-3 spreadsheet and the WordStar word processor that didn't make use of the bitmapped graphics anyway.

Next, Microsoft and IBM worked together on a product called IBM OS/2 Presentation Manager (Figure 31.6), which looked a lot like Windows, and Trower was tasked with creating Windows 2.0 in parallel with a separate team. A key goal was to make the interfaces be familiar to users of terminal-based mainframe applications, and they didn't have the freedom to design for the bitmap and mouse like applications on the Mac. Windows 2.0 was released in 1987, long before Presentation Manager, and was fully overlapped (Figure 31.7). It supported icons on the desktop, like the Star, and proportional fonts. Trower recounted that Steve Ballmer thought Windows would die off, and OS/2 would be the winner, but Bill Gates, who was negotiating with IBM, had a different view and continued the Windows development.

Trower left the Windows project after 2.0 but was still consulted on where Windows needed to go. Windows 3.0 from 1990 was able to have much better graphics

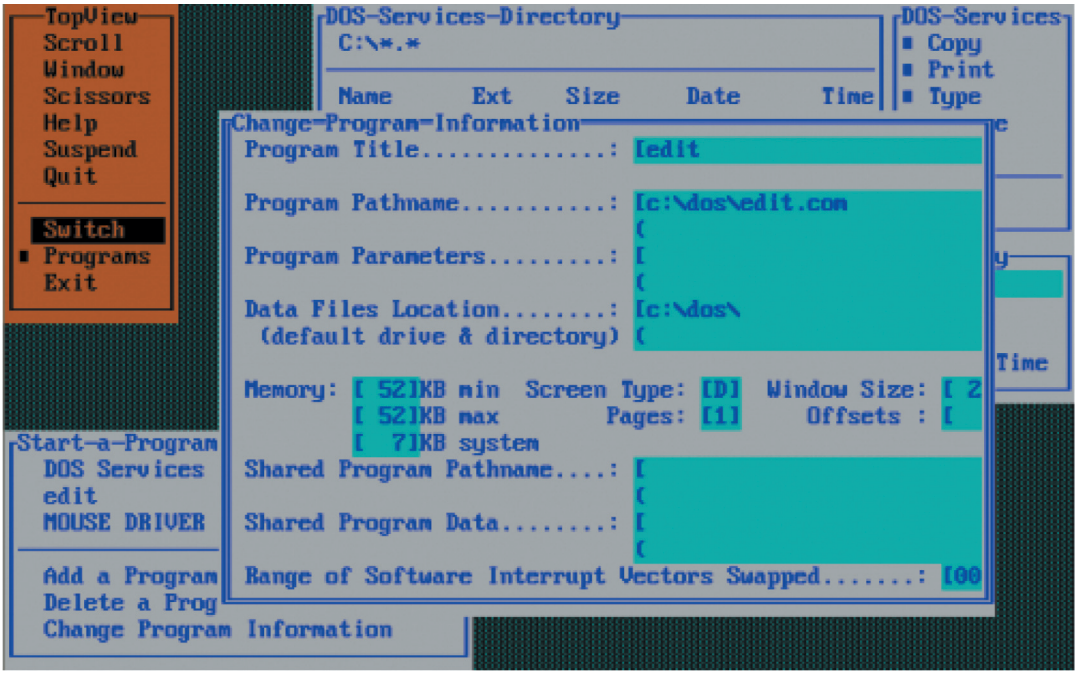


Figure 31.5 IBM's TopView window manager. Source: From Tandy Trower, reprinted with permission.

(taking advantage of the new hardware), longer, friendlier filenames, and added the "Program Manager" so users could collect and launch their applications from one place. He interfaced with IBM about user interface issues, but the competing OS products continued to be developed in parallel until eventually Microsoft disengaged from IBM's OS/2, which eventually was canceled.

31.4 Graphic Design

Microsoft did not have any graphic designers working on Windows 1.0, mainly because they were in such a rush. Also:

> I mean Windows was essentially in its first iteration, [and] was really designed to be an operating system. ... You didn't have a tremendous amount of pixels anyway on the screen, and so, ... the assumption was that, for the most part, you could just rely on the developers, which was a very bad idea, ... to design the icons.

Trower did bring in graphic designers with Windows 2.0, who turned out to be from the print industry. However,

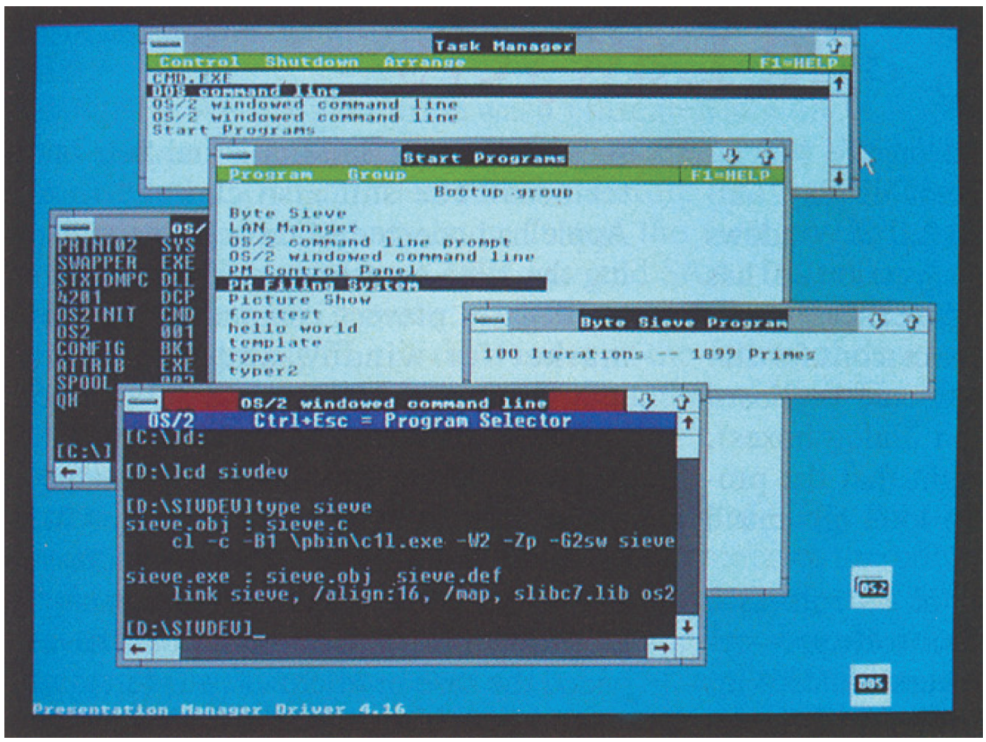


Figure 31.6 IBM and Microsoft worked together on OS/2 Presentation Manager. Source: From Tandy Trower, reprinted with permission.

> Bill Gates was very much a technical person himself. He could float between business plans and technical implementation details with ease. ... Bill always respected Steve [Jobs] for having a better sense of the aesthetics of what was important in the design of software products. And so early on that element was just missing. And it's odd that it was missing, because [Microsoft] had Xerox PARC people like Charles Simonyi, who was the architect of the early applications at Microsoft ..., and then there was ... Scott McGregor. Both these gentlemen understood the importance of visual design.

31.5 Microsoft's User Interface Design Group and Visual Basic

After releasing Windows 2.0, Trower was convinced that Microsoft needed its own usability labs in order to get back to the original goal of making systems more accessible to people. Inspired by HyperCard, the goal was to make it easier to build easy-to-use applications for Windows. He therefore formed and led the Microsoft User Interface Design Group (1988–1994) that developed Microsoft's user interface

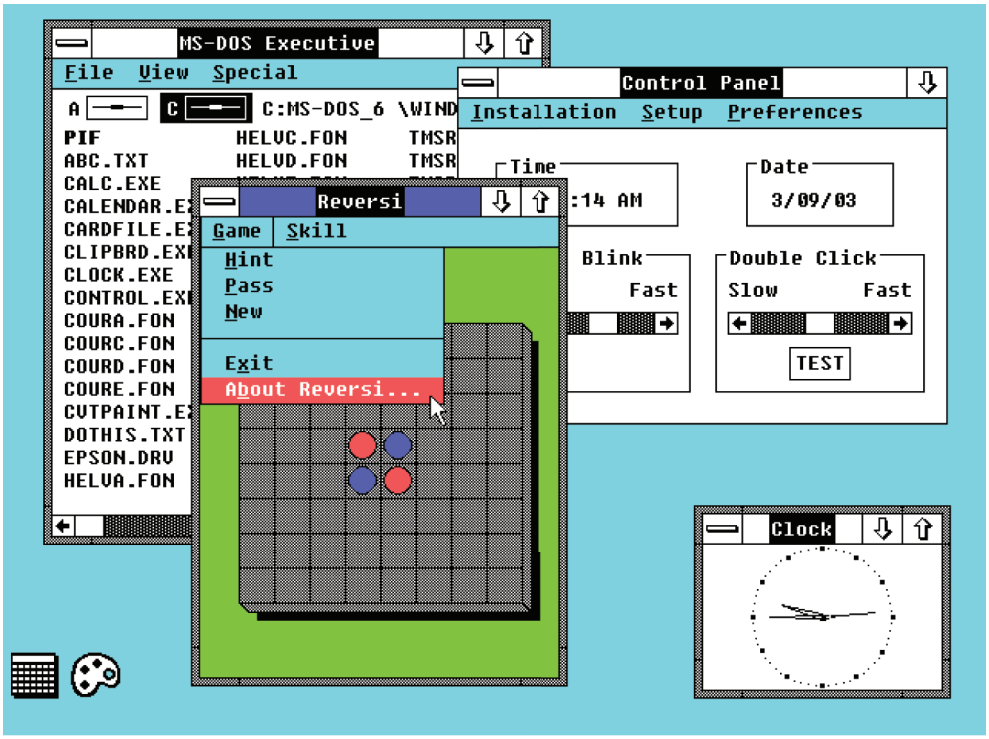


Figure 31.7 Windows 2.0 from 1987. Source: From Tandy Trower, reprinted with permission. Used with permission from Microsoft.

guidelines, provided access to experts in graphics design, performed usability testing, and investigated future UI concepts. He "also had a small team focused on designing what next-gen UI might look like (which ultimately was shown in Gates' 'Information at Your Fingertip' presentation)."

During that time, Trower said he "never lost my passion that Windows needed BASIC. So [he] personally worked on envisioning what that might be." However, the Visual Basic product was developed by others, and Trower served as an advisor and consultant on that project.

31.6 Windows NT and Windows 95

While Trower was in the UI Design Group, he became a "kind of in-house consultant" on the UIs for both Windows NT (code named "Cairo") and Windows 95 (code named "Chicago," see Figure 31.8)—two different operating systems by two different teams. The UIs looked basically the same.

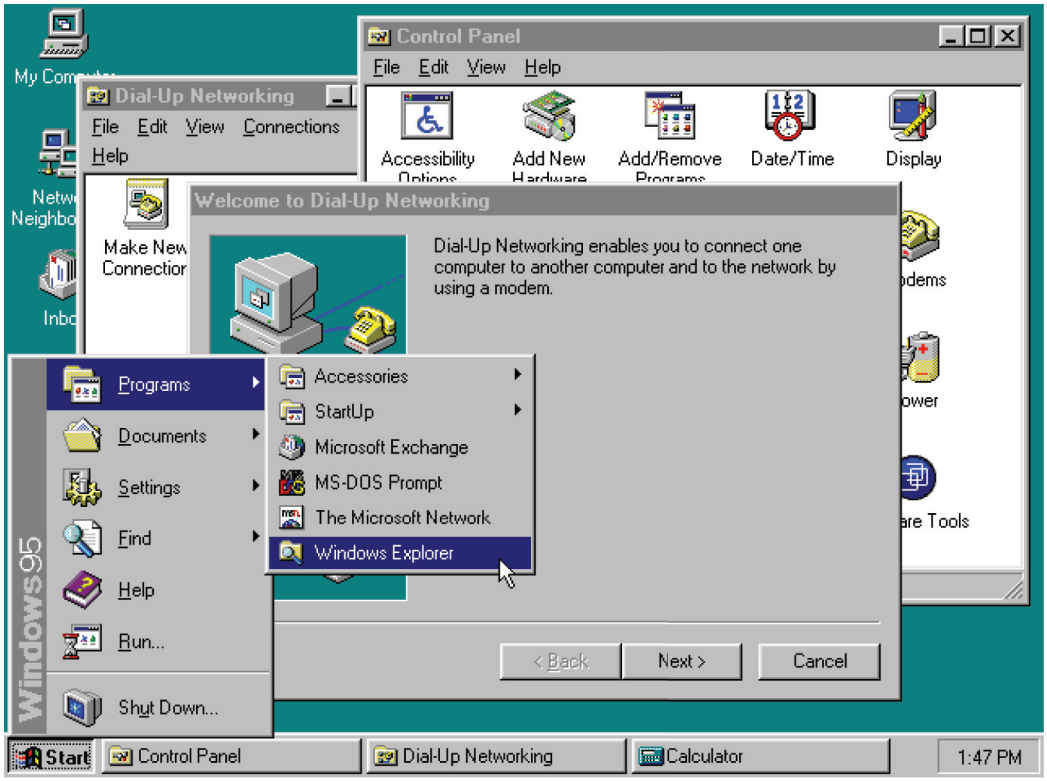


Figure 31.8 Windows 95. Source: From Tandy Trower, reprinted with permission. Used with permission from Microsoft.

> There was some good creative innovation that happened in this release and that moved beyond the program manager.... I worked with a team, and they created this new concept called the Start Menu, which exists in Windows even today, and the task bar, which allowed you to see ... applications ... at all times. In addition, this was the first release of Windows that included a web browser called Internet Explorer, and was also the first version of Windows that really considered how to set things up for people who ... have special needs .... And so, this was an incredibly important release.

31.7 Microsoft Bob and the Windows Assistant

Around this time, another project was creating Microsoft Bob (see Section 11.11.4), which aimed to be a friendlier way to interact with Windows and had "an active on-screen agent that would work with you." It was led by Melinda French who later

became Bill Gates's wife. It was "a major flop" and was withdrawn from the marketplace after only about six months. However, the agent architecture was used in the "Office Assistant," also colloquially known as "Clippy" (see Section 18.5). Trower was part of the team that built the software for the agent but:

> ...while I provided the fundamental technology, the Office group was not interested in my feedback about how to use this agent. And so they created a part of the interface that again was well-intentioned but unfortunately very much failed, and they later had to withdraw from the system.

31.8 Other Microsoft Products

Trower then left the interface group. "Most of the product groups by that time had created their own design teams. So that was good, so there was no need for a centralized team anymore." Trower worked on the Windows Media Center (2002), which was designed to allow users to watch TV on their computers—an idea much ahead of its time. The UI was designed to look very different from Windows (Figure 31.9). Meanwhile, Windows had a number of versions, with small tweaks to the UI, including Millennium and Windows XP.

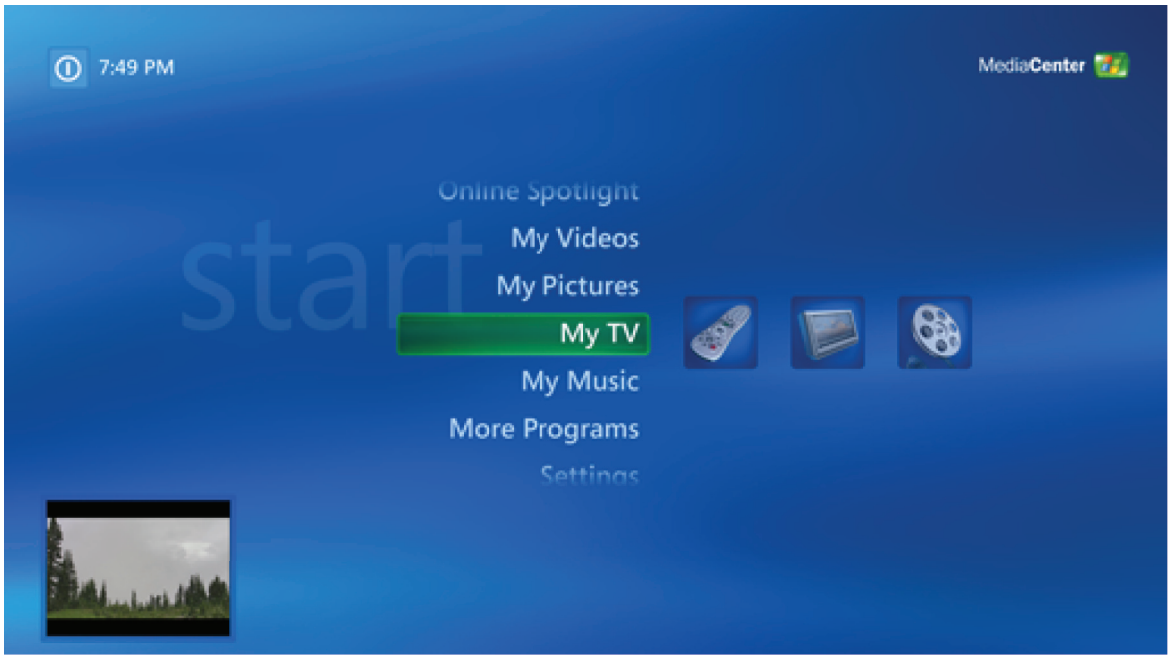


Figure 31.9 Windows Media Center (2002). Source: From Tandy Trower, reprinted with permission. Used with permission from Microsoft.

Then Windows 6 Vista (Figure 31.10) was released, which took much more advantage of advanced graphics, with transparency and images. Trower said that "Vista struggled in spite of the fact that it was visually very effective," and Windows 7 pretty much kept Vista's UI while improving the internals.

Figure 31.10 Windows 6 Vista (2006). Source: From Tandy Trower, reprinted with permission. Used with permission from Microsoft.

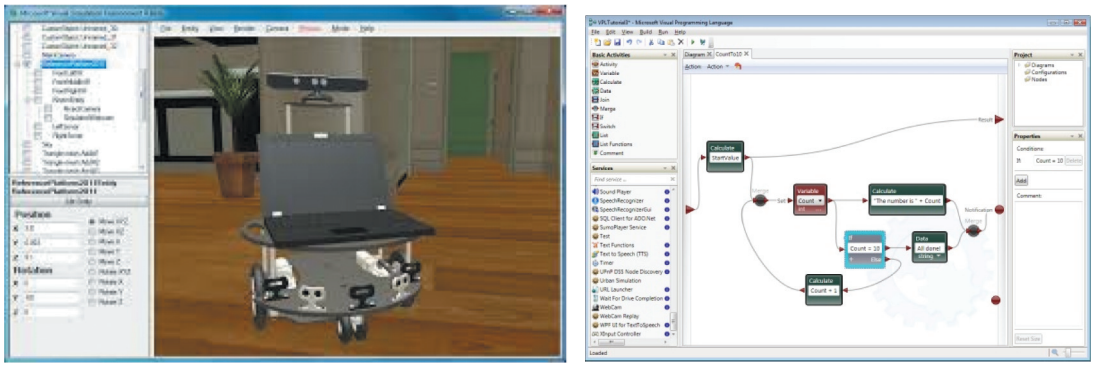


Figure 31.11 Microsoft Robotics Studio (2006). Source: From Tandy Trower, reprinted with permission. Used with permission from Microsoft.

Meanwhile, "The Office group after the failure of the Office Assistant decided that they still wanted to improve the overall interface, and so they kind of went to a fairly radical redesign of the overall application interface" and created the Toolbars and Ribbons in 2007 (see Section 9.5).

Then Trower started a group to create a common Robotics toolkit and visual programming language to try to make programming robots easier. The result was Microsoft Robotics Studio (2006) (Figure 31.11).

He left Microsoft in 2009 and founded a company called *Hoaloha Robotics* to make robots to help seniors and people with disabilities through more conversational interfaces.

CHAPTER 32

Interview: Loren Brichter

Figure 32.1 Loren Brichter. Source: From Loren Brichter, reprinted with permission.

Guest lecture from February 24, 2016.

32.1 Short Bio of Loren Brichter

Loren Brichter (Figure 32.1) was an engineer on the original iPhone at Apple around 2006–2007. He left in 2008 to start a company that made one of the most popular Twitter apps for iOS and Macintosh, called *Tweetie*, which won an Apple design award. It was ultimately sold to Twitter and became the official client. He has advised numerous companies and is the inventor on several patents, including the widely used *pull-to-refresh* interaction technique, developed while he was at Tweetie, which is now widely used on smartphones.

32.2 Pull-to-Refresh

Brichter founded his own company, Tweetie, to create a native iPhone app for Twitter. In 2009, Apple had just launched the App store and provided the application programming interfaces (APIs) to create iPhone apps. Brichter thought it would be a weekend project, but it:

> ...ended up dragging along for a few weeks, and it's important to understand that at the time nobody really knew what they were doing outside of Apple. People were designing really complicated junk. They still had the desktop mentality of having buttons everywhere. ... Apple was still fighting the good fight, trying to enforce the human interface guidelines. You could even tell then it was sort of a losing battle. ... But to me back then Apple was the absolute king of interaction design. Nobody could really touch them. So when I was writing this app, I was making a point to try and stick as closely as possible to the platform conventions they were trying to establish. ...So I ended up designing myself into a corner. ... The bulk of the screen was taken up by the timeline itself, and there was a title bar at the top [see Figure 32.2(a)]. ... You are allowed to have two buttons in the title bar. You know, [with] more than that, and it's just crazy. So I put the two most important buttons, and one was an account switcher, and the other was the compose-tweet button. ... I can't put the refresh button at the bottom. I can't put at the top because I'd already used both slots for buttons. I could have kind of squeezed it in, I mean technically there is room. But it was a pet peeve of mine to cram too much stuff up there, and I was trying to follow Apple's lead, and Apple would never, never, never have done that.
>
> So I had this realization. I was thinking, What is refreshing your timeline? What am I doing when I actually did it? I'm fetching new tweets. And it was just a way that Twitter works that those new tweets were guaranteed to be sorted at the top of the list.
>
> So if you have new tweets coming in and then we're going to go at the top of the list, why not put the button at the top of the list? [see Figure 32.2(b)] You know it kind of made sense. And there was even a handy API for this called `tableHeaderView`, which you know made it super easy to let me stick a chunk of view hierarchy at the top of a `tableView`. And the whole thing ended up being you know 15 or 20 lines of code, and I thought it was pretty clever.

Later, when Brichter was rewriting the app, he decided that it was not apparent enough how this feature worked, and that it needed better feedback so people

(a) (b)

Figure 32.2 (a) Tweetie main screen. (b) First design for pull-to-refresh. Source: From Loren Brichter, reprinted with permission.

would know how to operate it, since it was brand new. Fortunately, the Apple APIs provided a way to turn off clipping of the graphics outside of the scroll area, so he could display feedback and instructions above the content. Figure 32.3 shows the initial version of the feedback. When you pull down, you would scroll a little past the top of the content, and the first instruction would appear (Figure 32.3(a)). If you were scrolling, you couldn't help but see at least a little of the arrow due to the bounce at the end of scrolling (see also Section 5.9.1). When you pulled past a threshold, it would switch to the second one (Figure 32.3(b)). Then:

> ...at this point you had two options as the user. You, you can lift your finger, in which case the refresh would fire, the UI would change state, and you would get a loading indicator [Figure 32.3(c)]. Or, alternatively, you could keep your finger down and drag your finger back up. If you think of the UI widget as

(a) (b) (c)

Figure 32.3 Pull-to-refresh—first version of the feedback (stitched together from independent screenshots, which is why the dates are different). Source: From Loren Brichter, reprinted with permission.

> being armed, you could de-arm it and cancel the pending refresh; you have the choice of backing up and not firing it off.
>
> This idea of cancelability was super important to me. I felt that every good, general purpose, gesture had to have this property of cancelability. You know it's some wiggle room for human error.

He worked for a few more weeks to finish the Beta version of the Tweetie app, and he then sent it out to some trusted beta testers, without even mentioning the pull-to-refresh feature to see if people discovered it and could use it. The response was very positive:

> I got pretty good feedback from Beta testers without mentioning it at all ... and nobody seemed to have a problem finding it.... About a week before I was planning on releasing it, I started to get emails from Beta testers like this one: ... "I now find myself trying to pull down the list to refresh in mail. That's the sign of an apt gesture, I think." ... And I started to get a lot like that: "I just tried to refresh my inbox by pulling the list down and releasing it. I hate when you do things better than Apple." So I thought this was really funny.
>
> The gesture was so simple and so infectious that it was actually making other software on the platform feel broken in comparison.

One problem Brichter related was that he didn't do any formal user testing, and he didn't notice that if you held your phone in landscape mode, there almost wasn't enough room to pull your finger far enough down to pass the threshold since it was a fixed number of pixels.

Meanwhile, Facebook's app tried to use shaking the phone as a trigger for refresh, but this conflicted with Apple's shake-to-undo (see Section 15.3). Brichter related:

> a few weeks after the feature launched, there was a pretty decent open-source clone [of pull-to-refresh] that a lot of developers adopted and Facebook just dropped that into their UI framework that they were using for Facebook, and the Facebook app on iPhone then had it, and people loved it.

Other people also ported pull-to-refresh to Android, but it was a lot more difficult since Android does not support bouncing or pulling further than the top of the content (due to Apple asserting their scrolling patent, see Section 5.9.1). Eventually Apple bowed to public pressure and provided their own version of pull-to-refresh. All other apps had removed the refresh button, and people were trying to pull-to-refresh in Apple mail, but nothing happened.

> So there's this broken user expectation. And Apple took a really long time to do it.... I don't know what took them so long. Maybe it was not-invented-here syndrome, or something like that. But in IOS 6 they added the feature, and they added it as a UI kit level feature also, which meant it was part of the framework now. Any third-party developer could use it in their own apps, basically for free. You flip one Boolean in your code, and you have pull-to-refresh.

However, the Apple design was different than Brichter's. In the Apple version, as you pulled down, the refresh symbol would stretch (see Figure 32.4), and when you pulled far enough, it would start refreshing immediately. This "two-state" design was different from Brichter's three-state design (Figure 32.3). Brichter said "everybody called it the 'Booger'. This is the Booger refresh because you would stretch out this thing and it would snap," so there was less ability to cancel the operation.

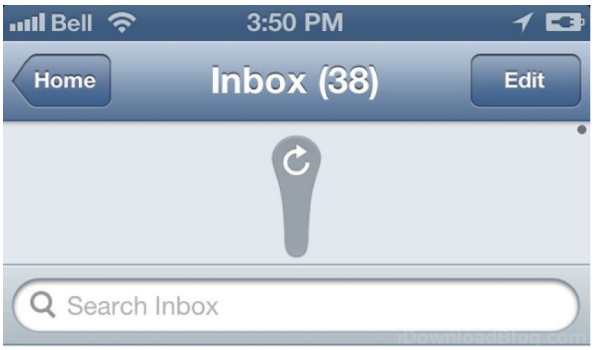


Figure 32.4 Apple's original pull-to-refresh. Source: From Loren Brichter, reprinted with permission.

32.3 Other Spring–Action Interactions

Brichter identified pull-to-refresh as an example of a "Spring–Action Interaction," where the user pulls an element like a spring and releases to do an action. He designed another spring–action interaction in the iPad version of Twitter, where if you pulled the last window to the side, a clear-all icon would be displayed, and if you kept pulling and released, then all your windows would be removed. Cancelability is really important in this case since it is a destructive operation.

Another spring–action interaction that Brichter really liked is the way you can pull email messages to the side to reveal action buttons, which appeared in 2013 in an email app called *Mailbox* that Brichter had nothing to do with. Eventually, Apple adopted this interaction for their own email program in iOS 8.

32.4 Pull-to-Refresh Patent

After Brichter released Tweetie 2, he was advised by his lawyer to submit a patent on the pull-to-refresh feature, which he did (US Patent 8,448,084 filed April 8, 2010). When he sold his company to Twitter, he stipulated, and Twitter agreed, that Twitter would only ever use the patent defensively, if someone sued Twitter first, so that Twitter would never sue anyone for using the pull-to-refresh feature.

CHAPTER 33

Interview: Don Hopkins

Figure 33.1 Don Hopkins. Source: From Don Hopkins, reprinted with permission.

Guest lecture from February 11, 2019.

33.1 Short Bio of Don Hopkins

Don Hopkins (Figure 33.1) was one of the original developers of Pie Menus, and he helped contribute to their popularity in games. He published a frequently cited paper about pie menus at the CHI'88 conference with John Raymond Callahan, Ben Shneiderman, and Mark Weiser [Callahan et al. 1988]. He then developed and refined pie menus for many platforms and applications including window managers, the Emacs text editors, universal remote controls, TV guide browsers, web browsers, visual programming interfaces, *SimCity*, and *The Sims*. These took advantage of many kinds of hardware including desktop, mobile, VR, OLPC, mouse,

stylus, and touch screens. He has published many free and open-source software implementations of pie menus for X10, X11, NeWS, Tcl/TK, ScriptX, ActiveX, OpenLaszlo, Python, JavaScript, C#, and Unity3D.

33.2 Definitions and Key Properties of Pie Menus

A pie menu is a circular menu (see also Section 7.3.9.1). Hopkins used the word "pie" to refer to the whole menu, and "slice" for one selectable element of the menu, with "item" as the label or content of that slice. He also used "target" for where the user does something (like click) to get the pie to appear. When making a pie menu, it is better for users to have an even number of slices, like 4 or 8, even if this means leaving empty slices (with no items in them).

A key property of a pie menu for Hopkins is that it must pop up with the center of the pie at the pointer location, so the user can move a tiny amount to get to the desired slice, and also so that the user can memorize which direction to move for each item and thereby provide efficient, eyes-free use. In his early mouse implementations, this included warping the cursor position to the menu center when it had to be moved away from the edge of the screen due to insufficient room.

33.3 History of Pie Menus

Hopkins wrote a Medium article on the "Pie menu timeline" [Hopkins 2019], which identifies Pixie from 1969 as the first place pie menus were discussed [Wiseman et al. 1969], also covered in the Newman textbook [Newman and Sproull 1973]. Hopkins reports that Lynn Bartram at the University of Waterloo was doing pie menus in 1984.

Hopkins started investigating pie menus, which he called Theta Menus, in 1986 at the University of Maryland with Mark Weiser, who was a professor there at the time. With encouragement from Ben Shneiderman, also faculty there, he created a version using the uwm window manager for the X10 windowing system on a Sun computer. An interesting challenge of that implementation was that X10 was very slow, so the user could easily mouse ahead, and the menus had to take that into account so it would perform all the correct actions. Hopkins made a new implementation of pie menus to support Jack Callahan's experiments comparing pie with linear menus, which were eventually published at the CHI'88 conference [Callahan et al. 1988]. In 1987, Hopkins implemented click-move-click behavior for pie menus since linear menus were already supporting that, in addition to press-drag-release (see also Section 7.3.11.3).

In 1997, Hopkins went to work at Maxis on *The Sims* game, which was released in March of 2000. Hopkins added pie menus for controlling the characters. The items in the menus were generated by the AI based on the context, the character's

Figure 33.2 Clicking on a character in *The Sims* popped up a dynamic pie menu, here with eight items, and the character's face in the center. Source: From Don Hopkins, reprinted with permission.

internal state, and who else is interacting with that character (e.g., the eight items in the menu of Figure 33.2). He also added pie menus to the *Multi Player SimCityNet* game for X11 on Linux, where six to eight square and rectangular icons would pop up around the cursor.

For the future, Hopkins advocates using pie menus in many more contexts, like for augmented reality/virtual reality.

CHAPTER 34

Interview: Ted Selker

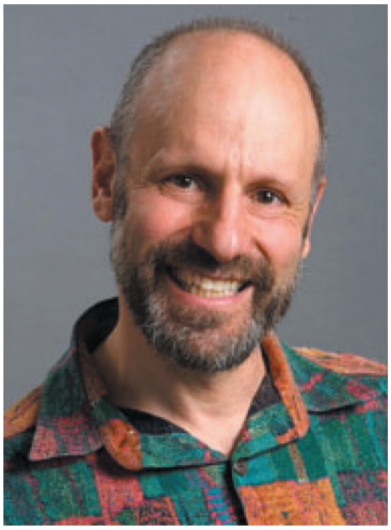


Figure 34.1 Ted Selker. Source: From Ted Selker, reprinted with permission.

Guest lecture from March 3, 2014.

34.1 Short Bio of Ted Selker

Ted Selker (Figure 34.1) is probably best known as the inventor of the TrackPoint in-keyboard pointing device while he was at IBM as an IBM Fellow and director of User Systems Ergonomics Research, but he has also invented a large number of other devices and interaction techniques. Selker spent five years as director of Considerate Systems research at Carnegie Mellon University Silicon Valley. He spent ten years as an associate professor at the MIT Media Laboratory where he created the Context Aware Computing group, co-directed the Caltech/MIT Voting Technology Project, and directed the CIDI Kitchen of the future/product design of the future project. Selker was co-recipient of the Computer Science Policy Leader

Award for Scientific American 50 in 2004, the American Association for People with Disabilities Thomas Paine Award for his work on voting technology in 2006, and the Telluride Tech fest award in 2008.

34.2 Origins of the Pointing Stick

Ted Selker said the inspiration for this work was the original 1980s papers studying the mouse by Card, Moran, Newell, and others (e.g., Card et al. [1980a, 1980b, 1983]) that showed the delay caused by needing to move your hands from the keyboard to the mouse. Selker felt that touchpads below the spacebar were not good because "it's very uncomfortable to use your thumb as a pointing device, even though your thumb is very sensitive and dexterous," and he observed that people often removed their hands from the keys with that kind of touchpad. So the goal was to create a pointing device that people could use without having to move their hands off the keyboard [Rutledge and Selker 1990]. In addition to the time penalty, using a mouse also requires the user to look away from the screen to get their hands positioned properly, and the mouse requires extra desk space which is sometimes not available.

The original idea was to put the pointing stick as part of the "J" key on the keyboard, so the user could wiggle the key up, down, left, or right to move the cursor, measured by strain gauges under the key. However, this required using a mode switch to tell the system whether you wanted to type a "J" or move the cursor, which caused a cognitive load and users forgot to be in the right mode.

Therefore, they decided to use a stick between the "G" and "H" keys since the G-to-H space is never traversed by a touch typist. They used a classic Fitts' Law test, with a target of varying size, moving around the screen, to test their pointing devices. The original design was slower than a mouse for purely pointing tasks but won when the times for the users to move their hand to the mouse from the keyboard (about 0.6 seconds) and back to the keyboard afterwards are counted [Rutledge and Selker 1990]. The pointing stick is a rate-controlled device that people had tried to perfect for decades. A previous paper reported that the biggest problem was overshooting, so they focused on high accuracy for the selection task. To improve the performance, they adjusted the transfer function (also called the gain function, see Section 4.6) to be non-linear (see Figure 34.2). They reported:

> We found that reducing sensitivity for near 0 forces allows users to do fine positioning more quickly. We were at first surprised to find that restricting the maximum speed of the cursor increased the overall [performance] speed. This allowed eye tracking and improved control. [Rutledge and Selker 1990]

Figure 34.2 Gain function used on an early pointing stick in 1990. Source: Screenshot from Rutledge and Selker [1990] at 6:18, reprinted with permission.

Selker explained that the plateau at the right of gain function in Figure 34.2 kept people from overshooting, and this speed is about how fast people's eyes can track the pointer so it doesn't get lost. However, people wanted to zoom around, especially to change contexts and change windows, so they added the "turbocharge" speed-up at the right edge. Selker's analogy was that people like to drive a "Ferrari ... but they crash it," so they only provided the turbocharge speed at the far right of the curve. In the end, this gain function resulted in "about 25% faster selection than any joystick algorithm that anybody had ever shown in the literature before." They also identified that people only have "about 5 bits of force control. That means one in 32 sections." This motivated keeping the transfer function flat for a number of values at the left side so that users would have more control.

Selker said that Stu Card presented a paper in 1990 arguing that a linear transfer function was always better, whereas Selker at the same conference presented a video, paper, and demo showing that a non-linear function was better. Selker said that he eventually tracked Card down in a mall with his laptop and showed him that the TrackPoint was better with the non-linear transfer function.

The original design for the pointing stick was a thin rod (Figure 34.3). In a visit to Japan, Selker met a researcher who argued that if the user felt the movement of the pointing device, it would be better, especially for the first 45s of use. So, Selker

Figure 34.3 Original pointing stick design was a rod. Source: Screenshot from Rutledge and Selker [1990] at 0:37, reprinted with permission.

tried many different materials for the top and finally used the material on a ping-pong paddle that has a little compliance so it would move a teeny bit. Selker said: "what we found which was very exciting was that while experts didn't improve, now, novices improved by 15% ... so that was a big, big deal."

34.3 Refining the TrackPoint

Now they needed to find a material for the tops that could be made at scale. "So the real trick was figuring out how to make a top that was grippy but not uncomfortable. The first ones (like Figure 34.3) were so uncomfortable that they put dents in your finger." As the first ones were going to market in a notebook computer in 1993, they wanted to use "plastic top with little bits of rubber in it" because it was cheaper, but that required users to press harder and potentially cause more repetitive strain injury. Furthermore, "about 10 days after the first products shipped in 1993, the [plastic] tops started breaking, because people were pressing on them so hard," and Selker was ready with a cheap rubber version that was very sticky. Unfortunately, that material absorbed oils from people's fingers, so they looked

for different materials and were inspired by the skin of certain fish from Japan that was quite sticky due to its structure.

Another story involved the wires inside the TrackPoint—the original version had long wires, but this was causing noise in the signals, and the initial devices at an initial trade show in New York City went faster to the left than the right. Eventually, Selker was able to redesign the entire system so that the TrackPoint was mounted on a small circuit board, eliminating the wires and reducing the cost from $12 down to $1.

They originally thought that putting the TrackPoint in regular keyboards would be useful for banks and other situations where having the extra room for a mouse was expensive, but this wasn't a big enough market for IBM at first. Where it took off was in the IBM ThinkPad laptops because the TrackPoint was a key competitive advantage. They later did sell keyboards with them and licensed the technology to other hardware companies like Toshiba, Zenith, Sony, HP, and so on. Dell's version made the buttons too stiff to press, which reduced the "ability to make selections by huge amounts."

The TrackPoint went to market in ThinkPad laptops, and everyone loved it, and the sales of ThinkPads took off. So IBM gave Selker more resources to continue to improve it. For example, they had optimized the TrackPoint for text selection so it was good at going horizontally and vertically, but you couldn't draw a circle with it. Ideally, it would be optimized differently depending on what the user was doing, but they couldn't change the low-level drivers and didn't have access to what the user was doing, so drawing remained a challenge.

Over the years there were many papers that compared the TrackPoint with other pointing devices, but Selker argues that many studies didn't give participants sufficient training time, and the TrackPoint is faster after sufficient practice (except for a mouse, where the TrackPoint is still better if you need to go back and forth from the keyboard).

Another story was that the algorithms they used were good for reducing the effects of tremor. This made it good for medical devices, where fine control was needed, and also for people with large tremors in their hands, like John McCarthy of Stanford. Blind people liked it because they could find it—it is always in one place, and people with physical disabilities found it usable because it was easier to control at low and high speeds. They experimented with putting TrackPoint into many kinds of devices, like remote controls, microphones for radiologists, and in a mouse instead of the scroll wheel. To make it more stable, Selker argued for there to be a wrist rest and larger buttons because the area below the spacebar, where the TrackPoint buttons are, was tiny in the original design.

Another experiment was to provide haptic feedback, where a piece of the TrackPoint would pop up against the finger, for example, when the cursor went across a window boundary. Blind people liked it, but it didn't help people after they were no longer novices, so it didn't go to market. He experimented with two pointing sticks in the same keyboard—he found that when they were asymmetrical, people could better remember the purpose for each. He explored using one for zoom and the other for select (Figure 34.4). What they did instead was to add a middle button (Figure 4.10), which goes into scroll mode. They added texture to this button so users can tell when they are on it without looking.

Figure 34.4 Experimenting with using two TrackPoints. Source: Ted Selker, reprinted with permission.

IBM tried bringing to market a mouse with a TrackPoint in the center instead of a scroll wheel, but Selker said that Microsoft removed the drivers that supported it, so the product failed.

Overall, it took about eight years of research to get the pointing stick to work, and it continued to be researched and refined. Selker said:

> And you'll see all of these stories, and trying all these things and it had to do with the fact that at every step when you're putting something new in product, people resent it, and they don't want it to happen, and you find yourself having to overcome even more obstacles.

CHAPTER 35

Interview: Shumin Zhai

Figure 35.1 Shumin Zhai. Source: From Shumin Zhai, reprinted with permission.

Guest lecture from Feb. 27, 2019.

35.1 Short Bio of Shumin Zhai

Shumin Zhai (Figure 35.1) is a principal scientist at Google where he leads and directs research, design, and development of human-device input methods and haptics systems. Previously, he was at IBM, where he invented the "ShapeWriter" text entry method for smartphones (which is a predecessor to Swype). At Google, he helped invent the Pixel phone squeeze interactions ("Active Edge"). In addition, his research career has contributed to both foundational models and understandings of HCI and practical user interface designs, inventions, and flagship products. His publications have won the ACM UIST Lasting Impact Award and the IEEE Computer Society Best Paper Award, among others. He was the editor-in-chief of the *ACM Transactions on Computer–Human Interaction* (2009–2015). He received his

Ph.D. degree from the University of Toronto in 1995. In 2006, he was named one of ACM's inaugural class of distinguished scientists. In 2010, he was named member of the CHI Academy and fellow of the ACM.

35.2 Effective Text Entry Requirements

Efficiency is a key requirement—so Shuman Zhai says, "you want to be able to use as little effort as possible to enter [text], with as little motor or perceptual or cognitive effort [as possible]." The other requirement is ease of learning and use—easy to begin using it, so it is easy to learn. Sometimes this is called being "natural," which Zhai says means it is "compatible or leveraging existing user skills." However, these properties are often in opposition.

What makes a product or interaction technique approachable or easy to use? Zhai says: "It's generally recognition-based, in psychological terms. That is, you recognize something, ... and then you select. It's 'outside-in' in terms of what's on the display [that] drives you to do the next [step]." However, efficient behavior means it is "recall-based or memory-driven behavior. It's 'inside-out', it really comes from their memory that drives their behavior. It's automatic ... that users are not highly aware of the details of the process." The question is how to design a UI that provides both features.

35.3 Origins of Shape Writing

The QWERTY keyboard was invented in the 1860s at the dawn of the Industrial Revolution to meet certain requirements like avoiding jamming and alternating the two hands, which don't apply today. However, we are stuck with this design since people are familiar with it, even though the layout was particularly bad for tapping on a virtual keyboard (Section 8.3.6) with a single stylus on the screens of PDAs like Palm Pilots. The stylus must travel back and forth across the keyboard repeatedly. Zhai's group first worked on changing the layout of the letters on the screen in the late 1990s and early 2000s, like the Metropolis keyboard (Figure 8.16(b)) [Zhai et al. 2000], but they found it was too hard to learn. The next idea was to use the shape of the gesture made when moving over the keys with the stylus. A patent on this idea (US 7,251,367 B2) was filed in 2002 and a paper appeared in 2003 [Zhai and Kristensson 2003]. The software was released by IBM alphaWorks in 2003, and a video was shown in the ACM UIST 2004 conference [Kristensson and Zhai 2004]. This version was called SHARK (see also Section 8.3.6.2). It works better on virtual keyboards with letters in an optimized layout, but these have proven too hard to get people to learn.

35.4 ShapeWriter

With the release of the iPhone in 2007, they saw an opportunity to popularize this method, so they spun off a company from IBM called ShapeWriter Inc. in 2008 that developed a WritingPad application. At the time, the iPhone did not support third-party keyboards, so all they could do was to have a separate app that had buttons to move the entered text to email or SMS. This app was released in the very first month that the iOS store opened. Zhai reported:

> And it was quickly... either 7th or 6th most downloaded app in the first month of iOS App Store. And it got some early adopters very excited about it. Here's one example, a comment from a reviewer user: "'revolutionized typing' is the understatement of the year. This technology should be part of every keyboard on all touch screens. Someone nominate these software developers for a Nobel. No joke!" And there are many comments like that, and *Time* magazine and major news outlets were also giving either awards or picking it as a featured app in their reviews.

ShapeWriter Inc. implemented it on all the platforms of the time, like Symbian and Windows Mobile, and today it is built into Apple iOS and Android. In a research project, Zhai ported the ShapeWriter technique to a smartwatch [Gordon et al. 2016] and got to over 20 wpm, compared to other watch text input techniques that require multiple taps per letter and that were only 10 wpm.

35.5 Algorithms

Recognition algorithms often use two components—a *spatial model* and a *language model*. The spatial model is used to decode the shape of the gesture—determining the probability of shapes to form each word. This shape is influenced by the users' pose—how they are holding a phone and which finger they are using. On average, this can be modeled with a standard Gaussian distribution formula, which results in "pretty simple computations at runtime." The language model predicts which words are likely to be typed. The simplest is just a word list. More complex models like "N-gram" take into account some number of previous words to predict the current or next word. This model can do best when one can limit the vocabulary, for example, by being domain-specific, and can be personalized to one person's usage, which has privacy concerns.

Zhai studied tapping on an on-screen keyboard [Fowler et al. 2015] and found that the word error rate when just using the exact position of the screen taps is 38.4%. But when taps are corrected using a language model, the error rate falls to 5.7%, and personalization can improve this further to 4.6% (Figure 35.2).

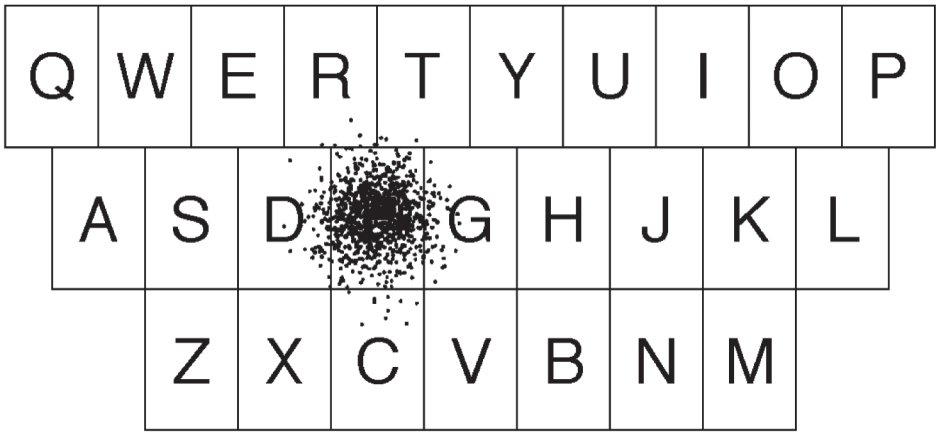


Figure 35.2 1,000 randomly sampled taps on the "F" key shows that many taps are on neighboring keys, many of which can be corrected by using a language model. Source: Fig. 1 of Fowler et al. [2015].

35.6 Pixel Squeeze-to-Assist in Google Pixel 2 Phone

There is a longstanding expectation that speech will dominate future interactions, so Google wanted to bring their Voice Assistant more prominently to their Google Pixel phone. Zhai's group figured out how "you just squeeze the whole phone, view it as an entire button, and you just squeeze the phone and then you start talking. ... But technically, it's actually surprisingly hard to do." The edges of the phone need to stay stiff, and the interaction needs to avoid false positives since users pick up the phone all the time, which has to be differentiated from an intentional squeeze. Their solution was described in a 2019 CHI paper [Quinn et al. 2019].

In conclusion, Zhai discussed what makes an invention successful, and it takes a *need*, *novelty* (a new way of approaching a problem domain), *know-how* (the technical/mathematical insight to implement a solution), *know-why* (the human factors insight of what makes the design useful), and a lot of *luck*.

CHAPTER 36

Interview: Gregg Vanderheiden

Figure 36.1 Gregg Vanderheiden. Source: Gregg Vanderheiden, reproduced with permission.

Guest lecture from March 31, 2022. This chapter is based on a summary created by Jae Kim, Jason Hsu, Tianying Lyu, and Simon Corpuz for class in Spring 2022, reused with permission.

36.1 Short Bio of Gregg Vanderheiden

Gregg Vanderheiden (Figure 36.1) is the founder and Director Emeritus of the Trace R&D Center and a professor in the College of Information Studies at the University

of Maryland. He has been active in the field of technology and disability for 50 years and was a pioneer in the field of Augmentative Communication (a term originating from his writings), assistive technology, and computer access. Access features developed by Dr. Vanderheiden and the Trace Center team can be found in every computer and mobile device internationally (Windows, MacOS, Linux, iOS, and Android), the Library of Congress talking book player, ATMs, voting machines and kiosks across the country, and more. He created the first World Wide Web Accessibility guidelines in 1995 and co-chaired the Working Group for WCAG 1 and 2. He is a past President and Fellow of the Rehabilitation Engineering and Assistive Technology Society of North America (RESNA), Founding Fellow of the American Institute of Medical and Biological Engineering (AIMBE), and a Fellow in the Human Factors and Ergonomics society.

36.2 Personal History

Gregg Vanderheiden first explained how he started off working in accessibility:

> This is my fiftieth year in technology and disability. I began in 1971 as an undergraduate student who was tricked into going out to a school to see a young lad who had cerebral palsy and could not speak or write, or communicate, except on a piece of wood that somebody had burned the letters of the alphabet into. So, I quit my job after meeting him, joined with [David Lamers] who tricked me into going out there, and we formed ... a group of undergraduates trying to find a solution for him, and that undergraduate volunteer sort of effort evolved into the Trace Center which then went on [and developed] nine of the first ten accessibility features in Windows [which] were licensed royalty-free from the Trace Center, the first accessibility features in Apple, the first Web guidelines in 1995, etc., etc. And the big long list of things that came out of it.

36.3 Digital Affinity

Vanderheiden uses the term *digital affinity* to describe one's ability to deal with technology. Digital affinity is a *talent* by nature. It is different from digital literacy, which is a skill that can be learned—"you can teach technology, but the ability for someone to absorb it is a function of talent." Digital affinity is also not equivalent to IQ. One can be very smart yet still not be able to use technology. And the "digital natives" who grow up with digital technologies and are fluent with search engines and smartphones are also not immune to this problem. It is like singing or athletics or artists—some have a talent for these and are natural at them, some are okay and can learn to be adequate, and others just can't do it at all.

Low digital affinity can have a big impact on people's ability to function:

> But if you can't use technology, it really affects almost any job. We have people that are flunking out of English in community colleges because they can't use the computers. ... And these are digital natives, right? And they can't use the technologies. So, what we need to realize is that for us techies all this is so simple. It doesn't occur to us that this is something that not everybody can do. It's going to be hard for others, but it's going to be out of reach for still others. ... We see it in community colleges. We see it in universities.

Vanderheiden mentions an example about watching a person who would wait for a long time in order to use a particular computer, when there were other identical computers available right next to it, because "I knew how to use this one." Another example was an English course in the community college where homework had to be turned in electronically. In the first several classes, the teacher had to go back and forth showing different students how to work on the homework on the computer, until by the third class she just gave up and gave everyone the markers and papers so they could finish by hand and scan it.

Accessibility features exist, but they exist in the control panel. For people with low digital affinity, going into the control panel is like asking people who aren't familiar with cars to go under the hood of the car and change something to make it work better for them.

Those with low digital affinity are frequently troubled by the thought that, "what if I break this?" Vanderheiden had met people at the job center who would lean over and squint at the screen, so his team made little cards to be put right next to the computer and teach people how to adjust size and contrast setting in the control panel. But some people are still afraid to follow because they don't want to break the computer, and even if they get help with more explanation, they might come back three days later feeling uncertain about themselves again. And these worries are well-founded—setting the controls incorrectly can indeed make the computer unusable, and there is no simple "Undo" for setting controls incorrectly. Thus, the controls that are supposed to allow people to customize their computers are in a place that is so scary that people won't use them.

And this is quite widespread:

> We were amazed at how many people really struggle with their computer. ... Many people who "can use computers" are only able to use them partially or superstitiously. Even a lot of our digital natives. ... Even in school they may be using iPads ... and then they get to get job training. They have to

> learn Windows or Mac, and they have no idea, because the operating systems are so completely different, and they just knew how to do things again sort of superstitiously. ... Why do you do it that way? Well that's just the way you do it.

And it continuously gets worse as technology gets more advanced. The iPhone and iPad became significantly more difficult to use when they took away the big physical home button (changing it to a gesture), and added many new features like split screen and multi-tasking.

36.4 Morphic

Morphic is a tool designed by Vanderheiden and an international group led by Vanderheiden and his team to make accessibility features easier to find and use (as well as other capabilities). Morphic includes a MorphicBar (a toolbar – see Figure 36.2) that provides shortcuts for many accessibility features including text size, magnifier, snip, read selected text, and contrast and color. The Windows system has read-aloud features in programs such as Word and Edge, but they all operate differently and can't work across programs. However, Morphic has capabilities built right in and can be activated or adjusted by a single click on the "Read Selected" buttons. Similarly, the text size buttons will increase the fonts in all programs and the desktop, not just in a browser.

With a right click on the Morphic bar, users can go to the control panel but are taken directly into where they need to go. In the directed position, they can pick a particular type of color blindness setting, adjust mouse or keyboard settings, select language preferences, and can change all other accessibility settings.

For universities, schools, libraries, companies and other organizations, Morphic also provides the ability to add three extra buttons that can be linked to many different things such as a university's printing service, content management application (like Canvas or Blackboard), email, or any other relevant features they want to have such as translation tools, Word Simplify (hiding or reorganizing the menu items), opening USB drives, and so on.

Once logged in, the user can ask Morphic to save all of their settings, including the settings of the computer and the assistive technologies. Then the user will be

Figure 36.2 A screenshot of the Morphic accessibility settings from Vanderheiden's presentation slides. Source: Gregg Vanderheiden, reproduced with permission.

able to log in and have instant, ultra-simple setup on all the computers in their home, library, training room, test center, or at grandma's house. It is particularly important for people with low resources if they are assistive technology (AT)—users that don't have their own laptop and have to move around using other computers at different locations. Morphic, with its AT-on-Demand feature, allows users to have their AT be automatically installed on any computer at a school, library, company, community center, and so on, that has Morphic on it – allowing the user's AT to follow them to any computer they encounter and need to use. Morphic can also adapt to the user's varying abilities during the day. There are people who may be fresh in the morning but get extremely tired by the end of the day, especially if they have physical disabilities, and Morphic can have different settings ready at different times. And for people with cognitive disabilities or very low digital affinity including many seniors, it can provide a simplified interface with a few specific buttons, like to join the family zoom call and to navigate to the family Google photo page. Morphic is also designed to stay the same even if the operating system or programs update and change their user interfaces.

A key feature is that when the user logs out of Morphic, it returns all the settings to their defaults, so people don't have to worry about messing up the computer for other people, and it is easier to share a computer among people with different needs.

36.5 Accessibility Today and Tomorrow

There has been great progress in accessibility to computers, especially compared to when Vanderheiden started in the 1970s, when there was none. However, there is still only "fractional coverage" since even though there are an "incredible number of features that it [are] built into [today's smartphones]," there is just a tiny fraction of people who could benefit who figure out how to use them because people are not aware of them, and they are too hard to set up. Other technologies have received less focus, and for cognitive disabilities, there is less support because it is really hard to know what to do when designing a computer that will work across the diverse range of cognitive, language, and learning disabilities. Adaptations can also be expensive, such as a $1,000 screen reader software for a $200 computer—who will pay for them?

For newer technologies, like virtual reality and augmented reality, and moving to gestural interfaces (like the reliance on swiping on smartphones), "it's just losing more and more people." The "next next generation interfaces" (NNGIs) of 20 years out, such as natural conversations, tactile displays, direct brain interfaces, and so on, will be more accessible to some – but less accessible to others. The guidelines for today won't cover all of these. If the innovations come from small companies,

then they won't have the knowledge or manpower to figure out how to make them accessible.

36.6 Info-bot

Vanderheiden's group is currently researching what they call an "Info-bot" that will leverage machine vision and AI so that it "would be able to perceive, understand, and operate any digital interface on any product that a median human user could." There would also be individual user interface generators (IUIGs) to generate a custom interface that matches individual abilities for any product encountered. Once a user finds the IUIG that best matches their condition, then anytime they walk up to any digital interface the Info-bot can look at it, figure it out, and pass back a description of the functionality along with the generated individual interface. This approach may have many potential advantages. For users, they can get almost ubiquity of accessibility for all products and personally consistent interfaces (as defined in Section 20.10). For developers, they don't have to make the devices be "accessible" for many kinds of disabilities following many guidelines; instead, they just need to make sure the Info-bot can understand the interfaces. For consumer advocates, they can just focus on figuring out what the interface should look like for the Info-bot. However, it will also require a new contract between industry, consumers, society, and government, but many aspects are already available in research labs.

36.7 Chapter Conclusions

Vanderheiden had some inspiring words for the students in the class:

> [B]ecause you are students, and sometimes as students, you sort of feel like you're trying to catch up, and that's true. You are trying to catch up, but you also can get out in front and lead. There's nothing that you can't do and there's just looking out, finding problems and solving them, can take you into an amazing journey that is extremely rewarding. And I'm sitting at the other end of one such journey and I can share with you the tremendous joy that comes from doing things that actually change lives and have impact on other people.

CHAPTER 37

Interview: Chris Harrison

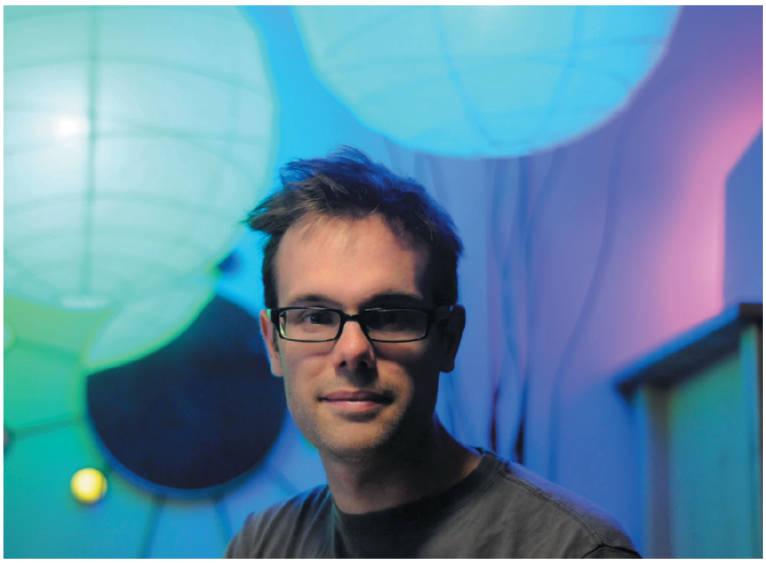


Figure 37.1 Chris Harrison. Source: From Chris Harrison, reprinted with permission.

Guest lecture from Apr. 9, 2014.

37.1 Short Bio of Chris Harrison

Chris Harrison (Figure 37.1) is an associate professor of Human–Computer Interaction at Carnegie Mellon University. He broadly investigates novel sensing technologies and interaction techniques, especially those that empower people to interact with small devices in big ways. Harrison was named as one of the top 30 scientists under 30 by Forbes, a top 35 innovator under 35 by MIT Technology Review, and one of six innovators to watch in 2013 by *Smithsonian*. Harrison has worked at Microsoft Research, IBM Research, AT&T Labs, and Disney Research, and he has been awarded Ph.D. fellowships by Google, Microsoft, and Qualcomm. When not

in the lab, Harrison can be found welding sculptures, blowing glass, and visiting remote corners of the globe.

37.2 Context

The Apple iPhone 5S was an amazingly fast computer. In fact, if you took that phone back to 1993, "it would be the third fastest computer on the planet, so it would have been classified as a supercomputer.... It's about 77 gigaflops of processing power." And the computer is a very small part of the physical phone device—the battery, screen, and antenna are all much bigger. But people haven't gotten smaller or faster. So, making devices smaller means making them "dumber" due to lack of room for the UI. And the physical size of devices (e.g., laptops, smartphones, and smartwatches) has stayed pretty much the same since around 2007. For the Apple Newton (1997), the electronics were the biggest element, bigger than the screen. A key challenge now is "how do we de-link size and capability?" Speech is a great example of something that works at any scale. Speech is fast, but it is highly linear and can't be used everywhere.

37.3 Using the Body for Input/Output

One idea Harrison's group has worked on is using the human body as input and output, such as projecting buttons onto a user's arm and hand and detecting where the person touches (see Figure 37.2) [Harrison et al. 2010b] or what gestures the user makes with their hands or other body parts. This is practical due to the incredible increase in brightness of tiny light sources like light-emitting diodes (LEDs).

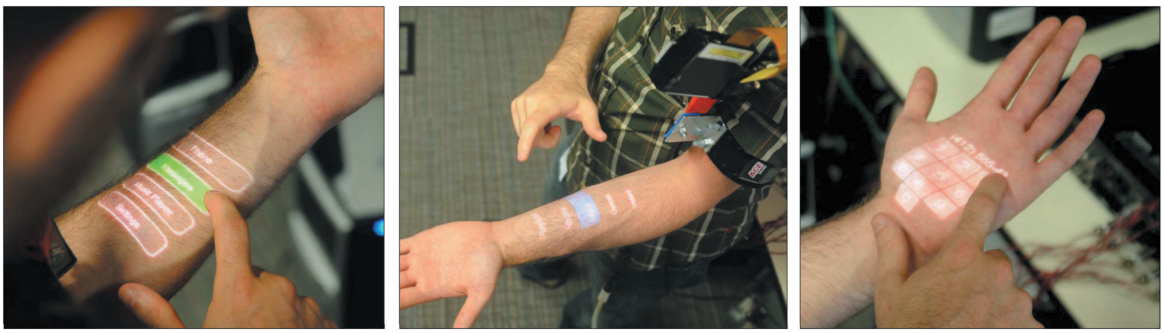


Figure 37.2 Mounting a projector and sensors onto a person's body. Source: Fig. 11 of Harrison et al. [2010b].

37.4 Increasing the Richness of Touch Interfaces

Another trend is how usability has increased with the evolution of devices—from no display, to text terminals, to GUIs and the mouse era, to the touch/mobile era, but the power of what users can do went *up* with each, until the mobile devices, when capabilities went *down* from desktop machines. It is much harder to write a document or edit a movie on a phone than on the desktop. The use of hands-on touch devices is pretty impoverished as well—for example, compared to using a guitar. For a phone, it is called a "two-finger swipe," but one doesn't say "three-finger handshake" in the real world—the number of fingers is not the important part. Fingers can actually do so many amazing things in the real world (and not just Pick, Click, and Flick!). So, we can do better, and there are many unused dimensions for touching.

One example that Harrison's group created, that he calls *TapSense*, uses the accelerometer to detect whether users are touching the device with their finger-pad, fingernail, or knuckle. These can be assigned to different actions, like using the knuckle to drag out a selection region versus using the tip of the finger for scrolling. Another dimension that can be detected is how hard you tap the screen or if you push toward a side while touching to create a shear force. These can be used to enable high-precision placement to match what can be done easily with a mouse. Harrison also figured out how to detect with current sensors what angle or pose the finger is touching the screen at. These features were commercialized as *Qeexo* [Xiao et al. 2015]. The touch sensors on devices could actually be sensitive enough to read a person's fingerprint or measure their body's capacitive profile, which can be used to enable collaborations, for example, on large screens, by knowing which finger goes with which person. The system could also detect hand poses, which might replace needing to use a "toolbar" or palette to select the digital "tool" to use.

Harrison has a long history of other investigations showing many other ways to use existing and new sensors to augment the usual interactions, and his group continues to invent interaction techniques that might be used for general or special-purpose interactions.

CHAPTER 38

Interview: Julia Schwarz

Figure 38.1 Julia Schwarz wearing a HoloLens 2. Source: From Julia Schwarz, reprinted with permission.

Guest lecture from Apr. 14, 2022.

38.1 Short Bio of Julia Schwarz

Julia Schwarz (Figure 38.1) is a software engineer who has worked on interaction techniques for touchscreens and augmented reality (AR) headsets since 2009. She received a B.S. from the University of Washington, and a Ph.D. from Carnegie Mellon University. Schwarz was named as one of the top 30 innovators under 30 by Forbes, and she has over 50 patents related to interaction techniques on touchscreens and AR devices. At Carnegie Mellon, she co-founded Qeexo (with Chris Harrison from Chapter 37), where she led the team that developed FingerSense,

a touchscreen technology that has shipped on hundreds of millions of Huawei phones. In 2015, she moved back home to Seattle to work on the HoloLens team at Microsoft. At Microsoft, she brought instinctual interactions to HoloLens 2, allowing users to directly interact with holograms using their hands.

38.2 Devices

Julia Schwarz works on the *Shell* team of the HoloLens, which is the "team that works on the Core UI of the operating system, which means the anything that shows up when you launch the HoloLens." She programmed the Shell software and created over 100 prototypes of example interaction techniques for the HoloLens 2 since "if you want to ship something in an industry, ... the best way ... is to build things and show it to people and show examples."

The HoloLens 1 was shipped in 2015, just after Schwarz joined Microsoft. It "is an augmented reality headset" that has a visor that you can see through and on which it can superimpose computer-generated images, and it also has outward facing cameras to see the world and the user's hands (see also Section 12.2). It is completely self-contained like a smartphone. The interaction model was "gaze, gesture, voice"—you move your head to position a cursor, and to select something you needed to make a specific pinching gesture called *airtap* with your fingers, or you can speak. This used the very accurate head tracking sensor on the HoloLens. However, "it's very difficult to teach this airtap pinching gesture. Most people can't figure out how to do it correctly. ... And tapping with your hand and aiming with your head is not immediately intuitive."

The HoloLens 2 was shipped in 2019. It has a larger field of view and:

> it's much more comfortable. You can actually wear it all day. In fact, that's how I develop—I just wear the HoloLens, and write code, flip down the visor to test, flip up the visor, and keep writing code.

It contains sensors to do head and motion tracking, a depth sensor aiming out to look for the user's hands, which means you need to keep your hands a little out front.

38.3 HoloLens 2 Interaction Model

The HoloLens 2 interaction model is completely different from HoloLens 1. The new device has:

> ... fully articulated hand tracking as well as eye tracking. ... You actually interact with holograms by touching them, which is sort of what most people expect to do when they see holograms. ... If you're far away from a hologram,

then you use a hand ray. ... [This is called] *instinctual interaction* because it's supposed to be easy to learn and easy to use.

38.4 HoloLens 2 Prototypes and Interactions

They mostly used Unity for programming interactions, but you can use Unreal Game Engine or directly coding in C++ using DirectX APIs for the HoloLens. Figure 38.2 shows the first prototype, which was just to reach out and pinch the corners of a cube to adjust the size. Key principles that Schwarz used, based on Don Norman's books, were to provide good feedback to users, like the corners lighting up pink and making a sound when grabbed. Another important property to making it feel compelling was that the corners followed the hand as closely as possible, without filtering or smoothing that would normally be used to remove jitter. A later prototype added a cursor, but only for interacting at a distance.

Schwarz worked closely with designers to make sure the prototypes looked nice and were understandable and clearly communicated. Together, they created a wide variety of 3D widget prototypes to test which were more natural. For example, for rotation, they tested grabbing a large knob with the hand, or grabbing a small handle on the knob with your fingers, or putting one finger in an indentation in the knob (Figure 38.3). Testing these prototypes with users was the key way they improved the UIs and decided what to put into the toolkit. For example, people

Figure 38.2 First prototype—pinching the corners of a cube. Source: From Julia Schwarz's video, reprinted with permission.

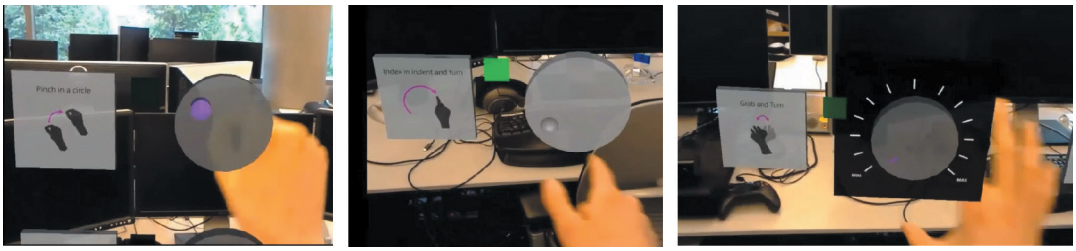


Figure 38.3 Three ways to rotate—pinch the purple handle, poke the indentation with a finger, or grab the whole knob with the hand. Source: From Julia Schwarz's video, reprinted with permission.

liked pinching handles and pushing buttons, but the lack of haptic feedback meant that people often accidentally moved off of the widget, which aborted the interaction prematurely. Therefore, they had to make the disengage volume significantly larger than the volume to select the widget, and they added exaggerated visual and audio feedback. Another issue is that some interactions resulted in the hands being in an orientation that was not "sensor-friendly" or that required that the low-level hand-recognition software be fixed to work more accurately for those poses.

To make the user tests more effective, they gave users tasks to do such as putting a virtual object into a specific position and orientation or even pointing at points on a circle like a Fitts' Law test (like Figure 4.24). This made it clear that while pushing an object was fun, it was less accurate than pinching and moving but could be sufficiently accurate if constrained, like to a one-dimensional movement or rotation. Without tasks, people liked all the techniques, but tasks made people reflect more on which *worked* best for them.

An interesting question is what to do when dragging a virtual object but the hands go out of view of the sensor. They considered dropping the object immediately or keeping it being dragged, which didn't work because users would end up with it stuck to their hands, so they dropped the object if the fingers were out of view for a few seconds.

People used different hand poses for gripping small versus large objects, but adding a small handle resulted in people naturally using an easier-to-detect pinching pose. It is also difficult to release objects accurately without having them accidentally move, so a more constrained movement, like pinching the corners or sides of a bounding cube is more precise. If objects move as if controlled by physics, this was fun, but for accuracy, it was necessary to add constraints.

Since Microsoft Windows has much 2D content, like the settings screens, they tested scrolling and clicking on 2D windows and found that it was very hard to tell

when the hand was intersecting the 2D plane of the window—so they added a visible cursor. For multi-finger interaction, like zooming, they required using the index fingers on two hands, but some people wanted one-handed pinch-to-zoom like on other devices, which was too hard to control due to lack of haptic feedback to indicate when the fingers were engaged. They provided a 2D keyboard that could also be controlled with the two index fingers. Typing with ten fingers was not reliable enough so that was left for future work.

The cursor was important to show where the finger was sensed with respect to the interaction plane for 2D interactions, and it also showed how far the hand is from the surface using a shadow. The cursor was shaped as a circle, with the selection point in the center, to avoid occluding the content. They provided a "contact pulse" as both audio and visual feedback when the finger hits the screen. Schwarz said:

> A story I like to tell is that kind of somewhat close to when we were announcing the HoloLens 2 we didn't have any sort of cursor implemented, and I just got so much feedback that people were planning to just cancel the whole idea of touching holograms and go back to HoloLens 1 because they thought hand tracking was completely broken, because they just could not navigate settings to like to set their IP address, ... and we just made one change, which was we turned on the cursor, and then I heard zero complaints. ... So when people aren't saying anything, that's actually a good sign. This also shows you the importance of this one detail, because the hand tracking didn't change, but we just added a cursor, and then people started to be able to successfully complete their tasks. And it's funny because people always think that what was wrong is the hand tracking, ... but sometimes it's just the visual design and the feedback.

They did a pose elicitation study of non-interactive hologram objects with novices and found that people treated them like their real-world counterparts. For example, they tried to hug a large house and grabbed other objects "in a million different ways" that would never work with their hand tracking. However, if they showed affordances like handles at the corners (like in Figure 38.2), then people naturally used them and even remembered to use the corners if the handles were later hidden. Providing *multiple* ways to interact with the objects proved too complex to implement since it couldn't tell which interaction was intended until it was started, which further motivated just providing the handles and pinching. They studied "micro-gestures," and the hand tracking was quite good for these, but using them in practice was challenging due to the problem of disengaging causing misrecognitions.

Interacting with objects *at a distance* is "the really hard thing, because there's no real-world analog.... And we tried many, many, many, many different interaction techniques ... [which] didn't work." So they ended up with a "hand ray" that goes from the shoulder to the palm since that proved to be the most stable and reliable approach. They tried using the index finger to point, but then there was no way to select since the pinch gesture moved the cursor. They next tried having a ray aligned with the user's wrist, which people found easy to point, but this also moved uncontrollably when pinching to select. Schwarz liked the idea of using "flying hands" (Figure 38.4), where copies of your hands fly to the object and then you can manipulate it as if it were nearby. This worked in highly constrained environments like a game, but in a general shell it was hard to know where the hands should fly to. In the end, they went with a ray from the shoulder to the hand, which was stable under pinch to select. The disadvantages are that it is fatiguing since it requires moving the arm, and it can be difficult to target items that are on the floor. This was considered to be the "least bad solution" they could find, and it is being adopted by other platforms like the Quest and Leap Motion.

38.5 Mixed Reality Toolkit

Microsoft provides a library called *Mixed Reality Toolkit* (MRTK) (http://aka.ms/mrtk) for hand interactions that can be used with the HoloLens or any other VR or AR device. It provides the interactions discussed above, including buttons, sliders, touching and moving virtual objects, and so on. It also provides a simulation mode using the mouse and keyboard, so the programmer can build everything without any special hardware, especially using the Unity interactive editor, and do rapid iterations.

Figure 38.4 Flying hands to interact at a distance. Source: From Julia Schwarz's video, reprinted with permission.

A cool feature for developers is that MRTK provides a "Holographic Remoting Player" so the application can be displayed directly on the HoloLens screen from the development computer rather than having to spend up to ten minutes compiling and downloading the code to the HoloLens itself.

Another feature is that MRTK supports connecting a Bluetooth mouse and keyboard to the HoloLens, which Schwarz mentions is good for rapid prototyping—for example, for switching modes or high-precision input—and can also be provided for end-users.

38.6 Chapter Conclusions

Julia Schwarz summarized her key takeaways as:

- Using visuals and sound to compensate for the lack of haptics for in-air interactions.
- Whenever possible, use familiar interactions like pressing buttons and pinching to grab objects.
- AR presents unique challenges compared to VR mainly because you can see your real hands and the scale and size of tasks are different in AR versus VR.
- Always be prototyping because it is much easier to influence people and disseminate ideas by showing them a working prototype. But you have to be open to their feedback.

In terms of the open challenges, Schwarz mentioned:

- Interacting with objects at a distance.
- Physics-based interactions, to allow for reliable grab and push interactions, especially in AR.
- "Low-calorie" interactions that reduce fatigue.
- Having better standardization to deal with buttons, sliders, and UX widgets for XR.
- Ways to increase the speed of iterating on prototypes.

PART V

FUTURE WORK AND CONCLUSIONS

CHAPTER 39

Future of Interaction Techniques

As of 2023 when I am writing this chapter, interactions for graphical user interfaces (GUIs) are highly evolved but surprisingly still familiar, with many being the same as they were when first explored in the 1960s. Still, there is plenty of work for future interaction technique designers and implementers. Many of these have been discussed in the chapters about specific techniques. Some of these have been investigated by students in my *Interaction Techniques* classes over the years.

39.1 IxT Class Projects

Many of the class projects involved creating new interaction techniques or better testing or documenting for existing ones. These may inspire further research or future class projects and illustrate how much work remains to be done in this space. This section lists some examples.

39.1.1 New Smartphone Interaction Techniques

One project group tried to improve the way users can move a text cursor on a phone given the difficulty of placing the cursor exactly on or between the target characters. Their idea was to use tilting of the device, but this proved twice as slow and much more error-prone than existing techniques. However, the students were able to improve their design by a factor of about two through user-centered iterations, reinforcing the importance of iterative design for IxT development.

Another group researched a gesture on a phone to search for the selected word, which (unsurprisingly) proved about 4.5 times faster than copy and pasting the word into a search field. Participants liked using a question mark gesture for this task. Another group looked at the task of reading on a smartphone and investigated special-purpose IxTs for navigation and annotations.

As smartphones were beginning to support foldable screens to provide a larger screen area, one group investigated what kind of folding would be preferred and what new interaction techniques might be based on the folding.

Another group created a special-purpose interaction technique for setting an alarm on the phone, which combined quick rough time selection with high-precision when needed, and integrated ancillary settings like the length of a snooze or multiple alarms at different intervals.

39.1.2 New Smartwatch Interaction Techniques

One device on which the conventional GUI interactions do not work well is clearly a smartwatch, and students have investigated various new interaction ideas. One project investigated different ways to scroll the content on a watch, with tilt scrolling testing poorly (which is not surprising given prior research, see Section 5.11.2 [Rahman et al. 2009, Xiao et al. 2014, Shima et al. 2016]), but sensing rubbing a user's thumb against the side of the index finger was popular and effective. Unfortunately, the students could only make this work by having the user wear a capacitive glove, but future sensors in a watch may make this unnecessary.

Two groups have tried to invent new smartwatch *text entry* techniques. The Watchboard used a physical keyboard intended to slide out from the watch band [Lockerman et al. 2017] (see Figure 8.19(b)), and a different group tried to use tilting and tapping on a watch for text entry, but both found their words-per-minute to be quite slow.

Another group investigated new ideas for using gestures detected by a smartwatch to transfer information from one physical device to another.

39.1.3 AR, VR, and 3D Interaction Techniques

A popular topic has been investigating interaction techniques for virtual reality (VR) and augmented reality (AR). For example, one group tested the Leap Motion device (see Figure 12.10(c)) and found it to be quite slow and inaccurate. Another group compared the Oculus controller to the Oculus camera-based hand tracking (where a selection ray emanated from the user's wrist) in a three-dimensional (3D) pointing test they created in Unity. Participants preferred using the controllers because they found the wrist-ray unnatural and frustrating. Participants also tended to move their heads to look at the targets even though they were all in view at the start.

One group investigated text entry in AR using a virtual full keyboard displayed on the user's forearm, a T9 keypad (see Section 8.3.3.2) on the user's non-dominant hand, and handwriting. Two different project groups looked at new ways that a

hierarchical menu of items could be presented in 3D. One group compared a vertical or horizontal ring, a semi-circle, a 3D "tulip" navigated by flick gestures, and a regular linear menu. Another group focused on presenting content in multiple layers, with partial transparency so the layers in the back were partially visible with the user moving around and selecting using gestures sensed by a Leap Motion device.

39.1.4 Wikipedia Entries

As mentioned in various places in this book, Wikipedia seems to mostly have unhelpful information about interaction techniques and their inventors, so many class projects over the years have created and edited Wikipedia entries. For example, groups have written new entries for several of the people who gave guest lectures and are featured in this book's interview chapters. Other groups have created and edited the entries for interaction techniques like scrolling and pull-to-refresh.

In the future, I hope this book can be used as a reference so many other Wikipedia entries about IxTs can be appropriately edited.

39.1.5 Comparison and Testing of Historical and Current Interaction Techniques

Several groups have tested existing interaction techniques using the standard methods described throughout this book. For example, one group tested the Contour RollerMouse Red *Plus* (Figure 4.23) and found it to be about 1/3 slower for novices to use compared to a mouse and, interestingly, about 17% slower for vertical movements compared to horizontal (the RollerMouse is fairly unique in using entirely different movements for the two directions, see Section 4.14.1). Another group tested various knobs, like the physical knobs on the TX-2 (Figure 2.8) and the dial on the original iPod (Figure 3.25), in comparison with two-finger scrolling and arrow keys, and they found that the iPod dial was actually fastest, with two-finger next, then the knob and arrow keys were the slowest. Another group analyzed 33 current scrolling techniques and devices, like flicking on various phones and tablets versus two-finger scroll on touchpads versus scroll wheels on a mouse, and found that the speed varied widely and depended on the task, with Page Up/Page Down keyboard keys (Figure 5.6(b)) being fastest for getting the desired screenful into view.

Before the chapters of this book existed, groups wrote summaries of specific existing interaction techniques. For example, to investigate differences in pointing, menus, text editing, and so on, one group did a side-by-side comparison of the interaction techniques on the Xerox Star compared to a current Apple iPhone. Another group collected a gallery of images of different IxTs of the same kind, like different menus. Another group described the interaction techniques they could

find that have been used on smartwatches, and yet another catalogued the VR devices and IxTs. Yet another group looked specifically at drawing programs and how well various interactions worked with a Wacom stylus and tablet, such as drawing a straight line or connecting dots.

Wendy Mackay and Michel Beaudouin-Lafon introduced the idea of having an *Interaction Museum* with implementations of old and new IxTs (see https://hci-museum.lri.fr/). Some class projects have tried to contribute to that by re-implementing some old techniques, such as the scrollbars from Bravo and Smalltalk, but none were of sufficient quality to be posted.

39.1.6 New Testing Methods

Multiple groups have looked into better ways to test scrolling interaction techniques since there hasn't been a standard way to do that. The latest results of that work are discussed in Section 5.11.1. Other groups looked into better ways to evaluate pointing, beyond the Fitts' Law tests (Section 4.14.1), but these were less successful.

39.1.7 Parts of This Book

Finally, a number of project participants helped author parts of this book, including providing summaries of the interviews, early proofreading, finding alternatives to copyrighted figures, summarizing some of the interviews for the start of those chapters, and researching Chinese text entry for Section 8.2.2.

39.2 Other Important Future Work

As mentioned throughout this book, there is important future work to be done for all kinds of interaction techniques. This section summarizes some of these.

39.2.1 UIs for Conversational Agents

While I am writing this (2023), the use of artificial intelligence (AI) in the UI is taking off exponentially. A key modality is using *chat*—text-based conversations, for example, with ChatGPT to create a conversation agent. The agent responds to the users' queries with a textual or graphical answer. These systems are also called *large language models* (LLMs), since that is the underlying technology, and *generative AI*, since they are often used to generate new text, pictures, or code. A key limitation currently is that they can produce inaccurate answers—for example, by "hallucinating facts" that aren't true.

In terms of the user interface for conversational agents, mostly the user types in natural language and gets back a static (non-interactive) answer, which the user can revise with further typing or by starting over with a different prompt (called

prompt engineering). What is missing is support for multi-modal user interfaces, which has been recognized as important since at least the 1980s (Section 18.4). For example, if ChatGPT is used to generate code, the user cannot currently point to the part that needs attention or use selection or widgets to adjust the results. The AI also does not have visibility into the state of UI widgets to enable users to get help with using complex applications [Ruoff et al. 2023].

39.2.2 Better Text Entry

As shown in Table 8.1, the speed for entering text into computers is still quite slow compared to how fast people can read. And the speeds for entering text into smartphones and smartwatches, as well as for 3D, VR, and AR, are especially bad. Although this has been an area of work for many years, there are still no good solutions. Speech recognition will undoubtedly improve, but there are many situations where that will not be appropriate or the best modality for users.

39.2.3 IxTs for 3D and Virtual and Augmented Reality

As shown in Chapters 12 and 38, much work is still needed for the interaction techniques for 3D user interfaces and especially for interfaces for VR and AR. Often, flat screens with conventional widgets will be displayed in the 3D world, which the user must try to poke at with their fingers or a controller. Today's interfaces are often fatiguing, requiring users to hold their hands and arms up in the air, along with being inaccurate, suffering from natural hand jitter and low-precision sensors. As these systems get to be more widely used (e.g., beyond games), more usable interaction techniques will be needed.

39.2.4 Two-Handed Interfaces

As Bill Buxton has been saying for many years (Chapter 25), too many of today's interactions with all devices only involve one hand. With the mouse or touchpad on a regular computer, to the touchscreen on a smartphone, tablet, or smartwatch, we are still only using one hand. Research systems have shown that interaction techniques leveraging both hands can be more effective [Buxton and Myers 1986, Zhai et al. 1997, Myers et al. 2000b], yet very few are in use. One promising exploration is to use two-handed interactions in IxTs for 3D, where users often have controllers in both hands anyway.

39.2.5 Brain Interfaces

Brain–computer interfaces, where the computer can get signals and potentially provide output directly with the user's brain, have been explored for years (Section 17.4.5) but still have a long way to go before they could be useful for people.

39.2.6 Other GUI IxTs

I argue that even "regular" interaction techniques could benefit from further research and improvements. Possibly the widget that has changed the most is the scrollbar (Figure 1.10), but recent versions are *less usable* than their predecessors, providing less information to users about where they are in the document and making it difficult to move long distances. Menu designs that were meant for up to a dozen choices now often provide hundreds or even thousands of options. Graphics editing for high resolution and complex designs now requires learning and choosing among a large number of interaction techniques and tools (for applications such as Photoshop and Illustrator). Text entry and editing on all devices should be faster and more accurate. Icons were created when users were likely to have under 100 documents, whereas today's computers and smartphones can store millions of photos, songs, and documents.

39.2.7 Open Data Model

We also need further work on how to *implement* user interfaces and interaction techniques. The current model (Chapter 20) has changed little since the 1980s and still focuses on an individual application's private data and its UI. The power of combining information *across* applications, which will be a natural way to interact with conversational agents, will not be possible unless the agent can leverage the data and capabilities of multiple apps. For example, one might want to combine information from the apps for Yelp, Maps, AMC, Cinemark, and a local movie chain to find a good place to eat after a movie is over rather than having to manually query all of these. I have called this vision the *Open Data Model* (Section 11.11.3), but alternative architectures like *plugins* have also been explored.

CHAPTER 40

Conclusions

Interaction techniques are fundamental to all user interfaces between people and technology. Besides being of interest to historians of technology, the different forms they have taken over the years help designers understand what the options are for designing and using interaction techniques effectively, and they help end-users understand all the nuances of how IxTs can be used. The stories about how they came to be and how inventors came up with the original designs are interesting and instructive, and many of these stories have never been told before.

Even if user interfaces change drastically in the future—for example, to be primarily mediated by natural language chat or voice—lower-level direct manipulation interaction techniques will still be necessary and important. After all, there are lots of buttons, knobs, levers, and visual displays with on-screen widgets on the starships *Enterprise* and *Millennium Falcon*!

It was great fun and very interesting researching and writing this book, and I hope you found it valuable. I am hopeful that future versions of this book can contain many new innovations in how people are interacting with their technology—new interaction techniques that are more usable and effective. Let me know how this book can be improved!

Bibliography

123test. n.d. Spatial reasoning test - free at 123test.com. Retrieved from https://www.123test.com/spatial-reasoning-test/.

T. F. Abraham. 2005. Background compilation in Visual Studio 2002, 2003 and 2005. Blog post. (December 9). http://www.tfabraham.com/blog/2005/12/background-compilation-in-visual-studio-2002-2003-and-2005/.

J. Accot and S. Zhai. 2002. More than dotting the I's—Foundations for crossing-based interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '02)*, Minneapolis, MI. ACM, 73–80. DOI: https://doi.org/10.1145/503376.503390.

Adobe Systems Inc. 1985. *PostScript Language Reference Manual*. Addison-Wesley.

C. Ahlberg and B. Shneiderman. 1994. The Alphaslider: A compact and rapid selector. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '94)*, Boston, MA. ACM, 365–371. DOI: https://doi.org/10.1145/191666.191790.

Y. Ahuvia. May 3. 2013. Infinite scroll: Let's get to the bottom of this. *Smashing Magazine*. https://www.smashingmagazine.com/2013/05/infinite-scrolling-lets-get-to-the-bottom-of-this/.

S. Al-khazraji, L. Berke, S. Kafle, P. Yeung, and M. Huenerfauth. 2018. Modeling the speed and timing of American Sign Language to generate realistic animations. In *Proceedings of the 20th International ACM SIGACCESS Conference on Computers and Accessibility (ASSETS '18)*, Galway, Ireland. ACM, 259–270. DOI: https://doi.org/10.1145/3234695.3236356.

S. R. Alpert. March. 1991. Self-describing animated icons for human–computer interaction: A research note. *Behav. Inf. Technol.* 10, 2, 149–152. DOI: https://doi.org/10.1080/01449299108924278.

C. Alvarado and R. Davis. 2007. Resolving ambiguities to create a natural computer-based sketching environment. In *ACM SIGGRAPH 2007 Courses*, San Diego, CA. ACM, 16–es. DOI: https://doi.org/10.1145/1281500.1281527.

J. R. Anderson, M. Matessa, and C. Lebiere. December. 1997. ACT-R: A theory of higher level cognition and its relation to visual attention. *Hum. Comput. Interact.* 12, 4, 439–462. DOI: https://doi.org/10.1207/s15327051hci1204_5.

ANS. 1984. *Draft Proposal American National Standard for the Functional Specification of the Programmer's Hierarchical Interactive Graphics Standard (PHIGS)*. American National Standards Committee X3H3/84-40.

Apollo Computer Inc. July. 1987. Open Dialog Interface Management System Supports IBM, DEC, and Sun. Product announcement.

C. Appert, O. Chapuis, and E. Pietriga. 2012. Dwell-and-spring: Undo for direct manipulation. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '12)*, Austin, TX. ACM, 1957–1966. DOI: https://doi.org/10.1145/2207676.2208339.

Apple. 1987. Knowledge navigator. In *ACM CHI 1992 Special Video Program*. *SIGGRAPH Video Review*, Issue 79. https://open-video.org/details.php?videoid=8129.

Apple Computer. 1992. *Macintosh Human Interface Guidelines*. Addison-Wesley, 418 pp. https://dl.acm.org/citation.cfm?id=573097.

Apple Inc. 2017. Download watchOS 3.0, 3.2.3 Information. July 19, 2017. https://support.apple.com/kb/DL1894?locale=en_US.

J. E. Archer Jr., R. Conway, and F. B. Schneider. 1984. User recovery and reversal in interactive systems. *ACM Trans. Program. Lang. Syst.* 6, 1, 1–19. DOI: https://doi.org/10.1145/357233.357234.

C. Asakawa and T. Itoh. 1998. User interface of a home page reader. In *Proceedings of the 3rd International ACM Conference on Assistive Technologies (ASSETS '98)*, Marina del Rey, CA. ACM, 149–156. DOI: https://doi.org/10.1145/274497.274526.

R. Baecker and A. Marcus. 1986. Design principles for the enhanced presentation of computer program source text. In M. Mantei and P. Orbeton (Eds.), *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '86)*, Boston. ACM, 51–58. DOI: https://doi.org/10.1145/22627.22348.

R. M. Baecker, D. Tilbrook, and M. Tuori. 1979. NEWSWHOLE—An interactive newspaper pagination system. *SIGGRAPH 79* (Republished in *SIGGRAPH Video Review*, 1, 1980). 10 minutes.

R. Balakrishnan, G. Fitzmaurice, G. Kurtenbach, and W. Buxton. 1999. Digital tape drawing. In *Proceedings of the 12th Annual ACM Symposium on User Interface Software and Technology (UIST '99)*, Asheville, NC. ACM, 161–169. DOI: http://dx.doi.org/10.1145/320719.322598.

E. Barrey. 1999. Methods, applications and limitations of gait analysis in horses. *Vet. J.* 157, 1, 7–22. DOI: http://dx.doi.org/10.1053/tvjl.1998.0297.

J. Barrilleaux. August 1. 2000. *3D User Interfaces with Java 3D*. Manning Publications.

L. Bass and P. Dewan (Eds.). 1993. *User Interface Software (Trends in Software)*. Wiley.

L. Bass and B. E. John. June 15. 2003. Linking usability to software architecture patterns through general scenarios. *J. Syst. Softw.* 66, 3, 187–197. https://www.sciencedirect.com/science/article/pii/S0164121202000766.

P. Baudisch, E. Cutrell, D. Robbins, M. Czerwinski, P. Tandler, B. Bederson, and A. Zierlinger. August. 2003. Drag-and-pop and drag-and-pick: Techniques for accessing remote screen content on touch- and pen-operated systems. In *Proceedings of Interact 2003*, Zurich Switzerland. IOS Press, 57–64.

M. D. Baum. June 9. 2008. Science & technology: Monkey uses brain power to feed itself with robotic arm. *Pitt Chronicle.* https://web.archive.org/web/20090910034547/http://www.chronicle.pitt.edu/?p=1478.

L. K. Baumeister, B. E. John, and M. D. Byrne. 2000. A comparison of tools for building GOMS models. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (SIGCHI '00)*, The Hague, Netherlands. ACM, 502–509. DOI: https://doi.org/10.1145/332040.332485.

P. Beardsley, J. van Baar, R. Raskar, and C. Forlines. 2005. Interaction using a handheld projector. *IEEE Comput. Graph. Appl.* 25, 1, 39–43. DOI: https://doi.org/10.1109/mcg.2005.12.

B. B. Bederson and J. D. Hollan. November. 1994. Pad++: A zooming graphical interface for exploring alternate interface physics. In *Proceedings of the 7th Annual ACM Symposium on User Interface Software and Technology (UIST '94)*, Marina del Rey, CA. ACM, 17–26. DOI: https://doi.org/10.1145/192426.192435.

B. B. Bederson and B. Shneiderman (Eds.). 2003. *The Craft of Information Visualization: Readings and Reflections*. Morgan Kaufmann.

B. Bederson, J. D. Hollan, K. Perlin, J. Meyer, D. Bacon, and G. Furnas. March. 1996. A zoomable graphical interface for exploring alternate interface physics. *J. Vis. Lang. Comput.* 7, 1, 3–31. DOI: https://doi.org/10.1006/jvlc.1996.0002.

B. B. Bederson, J. Meyer, and L. Good. November 6–8. 2000. Jazz: An extensible zoomable user interface graphics toolkit in Java. In *Proceedings of the 13th Annual ACM Symposium on User Interface Software and Technology (UIST '00)*, San Diego, CA. ACM, 171–180. DOI: https://doi.org/10.1145/354401.354754.

R. Bellamy, B. John, and S. Kogan. 2011. Deploying CogTool: Integrating quantitative usability assessment into real-world software development. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE '11)*, Honolulu, HI. ACM, 691–700. DOI: https://doi.org/10.1145/1985793.1985890.

E. Beneteau, O. K. Richards, M. R. Zhang, J. A. Kientz, J. C. Yip, and A. Hiniker. May. 2019. Communication breakdowns between families and Alexa. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems (CHI '19)*. Paper No. 243, 1–13. DOI: https://doi.org/10.1145/3290605.3300473.

E. Bergman (Ed.). 2000. *Information Appliances and Beyond*. Morgan Kaufmann, San Francisco.

E. Bergman and R. Haitani. 2000. Designing the PalmPilot: A conversation with Rob Haitani. In E. Bergman (Ed.), *Information Appliances and Beyond*. Morgan Kaufmann, San Francisco, 82–102.

T. Berlage. September. 1994. A selective undo mechanism for graphical user interfaces based on command objects. *ACM Trans. Comput. Hum. Interact.* 1, 3, 269–294. DOI: https://doi.org/10.1145/196699.196721.

T. Berlage and A. Genau. September 20–22. 1993. From undo to multi-user applications. In *Vienna Conference (VCHCI '93)*, Vienna, Austria, Vol. 733: Lecture Notes in Computer Science. Springer, 213–224. DOI: https://doi.org/10.1007/3-540-57312-7_70.

T. Berners-Lee. November 8. 1993. *HTML+ (Hypertext Markup Format)*. https://www.w3.org/MarkUp/HTMLPlus/htmlplus_1.html.

T. Berners-Lee, J. Hendler, and O. Lassila. May. 2001. The semantic web. *Sci. Am.* 284, 5, 34–43.

W. L. Bewley, T. L. Roberts, D. Schroit, and W. L. Verplank. December. 1983. Human factors testing in the design of Xerox's 8010 "Star" office workstation. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '83)*, Boston, MA. ACM, 72–77. DOI: https://doi.org/10.1145/800045.801584.

H. Beyer and K. Holtzblatt. 1998. *Contextual Design: Defining Custom-Centered Systems*. Morgan Kaufmann, San Francisco, CA.

E. A. Bier and M. C. Stone. August. 1986. Snap-dragging. In *Proceedings SIGGRAPH'86 Computer Graphics*. Dallas, TX. ACM, 233–240. DOI: https://doi.org/10.1145/15922.15912.

E. A. Bier and S. Freeman. November. 1991. MMM: A user interface architecture for shared editors on a single screen. In *Proceedings of the 4th Annual ACM Symposium on User Interface Software and Technology (UIST '91)*, Hilton Head, SC. ACM, 79–86. DOI: https://doi.org/10.1145/120782.120791.

E. A. Bier, S. Freeman, and K. Pier. 1992. MMM: The multi-device multi-user multi-editor. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '92)*, Monterey, CA. ACM, 645–646. DOI: https://doi.org/10.1145/142750.143065.

E. A. Bier, M. C. Stone, K. Pier, W. Buxton, and T. D. DeRose. 1993. Toolglass and magic lenses: The see-through interface. In *Computer Graphics, Proceedings SIGGRAPH'93*. 73–80. DOI: https://doi.org/10.1145/166117.166126.

J. P. Bigham and A. C. Cavender. 2009. Evaluating existing audio CAPTCHAs and an interface optimized for non-visual use. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '09)*, Boston, MA. ACM, 1829–1838. DOI: https://doi.org/10.1145/1518701.1518983.

D. Bledsoe Jr. 2011. Handwriting speed in an adult population. *Adv. Occup. Ther. Pract*. 27, 22, 10.

S. A. Bly and J. K. Rosenberg. April. 1986. A comparison of tiled and overlapping windows. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '86)*, Boston, MA. ACM, 101–106. DOI: https://doi.org/10.1145/22627.22356.

D. Bohn. September 30. 2019. Google's Project Jacquard is available on new Levi's jackets. *The Verge*. https://www.theverge.com/2019/9/30/20888909/google-levi-project-jacquard-available-trucker-sherpa-jackets-price-design.

R. A. Bolt. 1979. *Spatial Data-Management*. Architecture Machine Group, Massachusetts Institute of Technology, Cambridge, MA. https://www.media.mit.edu/speech/papers/1979/bolt_1979_spatial_data-management.pdf.

R. A. Bolt. July. 1980. "Put-that-there": Voice and gesture at the graphics interface. *ACM SIGGRAPH Comput. Graph*. 14, 3, 262–270. DOI: https://doi.org/10.1145/965105.807503.

G.-P. Bonneau, T. Ertl, and G. M. Nielson. 2006. *Scientific Visualization: The Visual Extraction of Knowledge from Data*. Birkhäuser.

T. Bonura, J. R. Miller, B. Nardi, and D. Wright. 1996. System and method for performing an action on a structure in computer-generated data. USA Patent No. 5946647, Filed Feb. 1, 1996, Issued Aug. 31, 1999.

A. Borning and R. Duisberg. October. 1986. Constraint-based tools for building user interfaces. *ACM Trans. Graph.* 5, 4, 345–374. DOI: https://doi.org/10.1145/27623.29354.

B. Bosker. 2010. Visual guide to Facebook's privacy changes over time. *The Huffington Post.* July 7, 2010. https://www.huffpost.com/entry/facebook-privacy-changes_n_568345.

A. Bragdon, S. P. Reiss, R. Zeleznik, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and J. J. LaViola. 2010. Code bubbles: Rethinking the user interface paradigm of integrated development environments. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, ICSE '10*. ACM, 455–464. DOI: https://doi.org/10.1145/1806799.1806866.

J. Brooke. 1996. SUS: A "quick and dirty" usability scale. In P. W. Jordan, B. Thomas, I. L. McClelland, and B. Weerdmeester (Eds.), *Usability Evaluation in Industry.* Taylor & Francis, London, 189–194.

F. Brooks. 1977. The computer "scientist" as toolsmith—Studies in interactive computer graphics. In *IFIP Conference Proceedings*. North-Holland, 625–634.

K. J. Brooks. 2022. BlackBerry pulls the plug on its pioneering mobile phone. *CBS News*. January 5, 2022. https://www.cbsnews.com/news/blackberry-devices-shut-down-service/.

D. Bryant and A. Howard. 2019. A comparative analysis of emotion-detecting AI systems with respect to algorithm performance and dataset diversity. In *Proceedings of the 2019 AAAI/ACM Conference on AI, Ethics, and Society (AIES '19)*, Honolulu, HI. ACM, 377–382. DOI: https://doi.org/10.1145/3306618.3314284.

M. Brysbaert. December 1. 2019. How many words do we read per minute? A review and meta-analysis of reading rate. *J. Mem. Lang.* 109, 104047, 1–30. DOI: https://doi.org/10.31234/osf.io/xynwg, https://www.sciencedirect.com/science/article/pii/S0749596X19300786.

E. Buehler, A. Hurst, and M. Hofmann. 2014. Coming to grips: 3D printing for accessibility. In *Proceedings of the 16th International ACM SIGACCESS Conference on Computers & Accessibility (ASSETS '14)*, Rochester, NY. ACM, 291–292. DOI: https://doi.org/10.1145/2661334.2661345.

V. Bush. July. 1945. As we may think. *Atlantic* 176, 1, 101–108. Reprinted and discussed in *Interactions*, 3, 2, March 1996, 35–46. DOI: https://doi.org/10.1145/227181.227186.

W. Buxton. May. 1982. An informal study of selection positioning tasks. In *Proceedings of Graphics Interface '82, the 8th Conference of the Canadian Man–Computer Communications Society*, Toronto, ON. National Research Council of Canada, 23–328. DOI: https://doi.org/10.20380/GI1982.47. http://www.billbuxton.com/SelecPos.html.

W. Buxton. January. 1983. Lexical and pragmatic considerations of input structures. *Comput. Graph.* 17, 1, 31–37. DOI: https://doi.org/10.1145/988584.988586.

W. Buxton. 1990. A three-state model of graphical input. In *Human-Computer Interaction – INTERACT '90*. Elsevier Science Publishers B.V., North-Holland, Amsterdam, 449–456.

B. Buxton. 2007. *Sketching User Experiences: Getting the Design Right and the Right Design*. Morgan Kaufmann, San Francisco, CA.

W. Buxton and B. Myers. April. 1986. A study in two-handed input. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '86)*, Boston, MA. ACM, 321–326. DOI: https://doi.org/10.1145/22627.22390.

W. Buxton, R. Sniderman, W. Reeves, S. Patel, and R. Baecker. 1979. The evolution of the SSSP score editing tools. *Comput. Music J.* 3, 4, 14–60.

W. Buxton, E. Fiume, R. Hill, A. Lee, and C. Woo. 1983a. Continuous hand-gesture driven input. In *Proceedings Graphics Interface (GI '83)*. National Research Council of Canada, 191–195. DOI: https://doi.org/10.20380/GI1983.31.

W. Buxton, M. R. Lamb, D. Sherman, and K. C. Smith. July. 1983b. Towards a comprehensive user interface management system. *Comput. Graph.* 17, 3, 35–42. DOI: https://doi.org/10.1145/800059.801130.

W. Buxton, R. Baecker, W. Clark, F. Richardson, I. Sutherland, W. R. B. Sutherland, and A. Henderson. 2005. Interaction at Lincoln Laboratory in the 1960's: Looking forward—Looking back. In *CHI '05 Extended Abstracts on Human Factors in Computing Systems (CHI EA '05)*, Portland, OR. ACM, 1162–1167. DOI: https://doi.org/10.1145/1056808.1056864.

P. R. Calder and M. A. Linton. October. 1990. Glyphs: Flyweight objects for user interfaces. In *Proceedings of the 3rd Annual ACM SIGGRAPH Symposium on User Interface Software and Technology (UIST '90)*, Snowbird, Utah. ACM, 92–101. DOI: https://doi.org/10.1145/97924.97935.

J. Callahan, D. Hopkins, M. Weiser, and B. Shneiderman. 1988. An empirical comparison of pie vs. linear menus. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '88)*, Washington, DC. ACM, 95–100. DOI: https://doi.org/10.1145/57167.57182.

X. Cao, C. Forlines, and R. Balakrishnan. 2007. Multi-user interaction using handheld projectors. In *Proceedings of the 20th Annual ACM Symposium on User Interface Software and Technology (UIST '07)*, Newport, RI. ACM, 43–52. DOI: https://doi.org/10.1145/1294211.1294220.

S. P. Capps, E. H. Beernink, J. R. Meier, and D. Temkin. 1993. Method and apparatus for correcting words. USA Patent No. 5367453, Filed Aug. 2, 1993, Issued Nov. 22, 1994.

S. K. Card and A. Henderson. 1987. A multiple virtual-workspace interface to support user task switching. In *Proceedings of the SIGCHI/GI Conference on Human Factors in Computing Systems and Graphics Interface (CHI '87)*, Toronto, ON. ACM, 53–59. DOI: https://doi.org/10.1145/29933.30860.

S. K. Card, W. K. English, and B. J. Burr. August 1. 1978. Evaluation of mouse, rate-controlled isometric joystick, step keys, and text keys for text selection on a CRT. *Ergonomics* 21, 8, 601–613. DOI: https://doi.org/10.1080/00140137808931762.

S. K. Card, T. P. Moran, and A. Newell. July. 1980a. The keystroke-level model for user performance time with interactive systems. *Commun. ACM* 23, 7, 396–410. DOI: https://doi.org/10.1145/358886.358895.

S. K. Card, T. P. Moran, and A. Newell. 1980b. Computer text-editing: An information-processing analysis of a routine cognitive skill. *Cogn. Psychol.* 12, 1, 32–74. DOI: https://doi.org/10.1016/0010-0285(80)90003-1, http://www.sciencedirect.com/science/article/pii/0010028580900031.

S. K. Card, T. P. Moran, and A. Newell. 1983. *The Psychology of Human–Computer Interaction*. Lawrence Erlbaum Associates, Hillsdale, NJ.

S. K. Card, J. D. Mackinlay, and G. G. Robertson. April. 1990. The design space of input devices. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '90)*, Seattle, WA. ACM, 117–124. DOI: https://doi.org/10.1145/97243.97263.

S. K. Card, J. D. Mackinlay, and B. Shneiderman. 1999. *Readings in Information Visualization: Using Vision to Think*. Morgan Kaufmann, 686 pp.

L. Cardelli. October. 1988. Building user interfaces by direct manipulation. In *Proceedings of the 1st Annual ACM SIGGRAPH Symposium on User Interface Software (UIST '88)*, Banff, AB. ACM, 152–166. DOI: https://doi.org/10.1145/62402.62428.

G. Casiez, D. Vogel, R. Balakrishnan, and A. Cockburn. 2008. The impact of control-display gain on user performance in pointing tasks. *Hum. Comput. Interact.* 23, 3, 215–250. DOI: http://dx.doi.org/10.1080/07370020802278163.

CDC, National Center on Birth Defects and Developmental Disabilities—Centers for Disease Control and Prevention. 2020a. Disability Impacts All of Us. https://www.cdc.gov/ncbddd/disabilityandhealth/infographic-disability-impacts-all.html.

CDC, Vision Health Initiative—Centers for Disease Control and Prevention. June 9, 2020b. Fast Facts of Common Eye Disorders. https://www.cdc.gov/visionhealth/basics/ced/fastfacts.htm.

K. S.-P. Chang, B. A. Myers, G. M. Cahill, S. Simanta, E. Morris, and G. Lewis. 2013. Improving structured data entry on mobile devices. In *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology (UIST '13)*, St. Andrews, Scotland. ACM, 75–84. DOI: https://doi.org/10.1145/2501988.2502043.

D. H. Chau, B. Myers, and A. Faulring. 2008. What to do when search fails: Finding information by association. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '08)*, Florence, Italy. ACM, 999–1008. DOI: https://doi.org/10.1145/1357054.1357208.

I. Chaudhri, B. Ording, F. A. Anzures, M. van Os, S. O. Lemay, S. Forstall, and G. Christie. 2010. Unlocking a device by performing gestures on an unlock image. USA Patent No. 7657849 B2, Filed Dec. 23, 2005, Issued Feb. 2, 2010.

M. Chen, S. J. Mountford, and A. Sellen. 1988. A study in interactive 3-D rotation using 2-D control devices. In *Proceedings of the 15th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '88)*. ACM, 121–129. DOI: https://doi.org/10.1145/54852.378497.

C. Chen, B. A. Myers, C. Ergin, E. Porat, S. Li, and C. Wang. 2023. ScrollTest: Evaluating scrolling speed and accuracy. Submitted for Publication. arXiv:2210.00735 [cs.HC] (preprint). https://arxiv.org/abs/2210.00735.

R. Chimera. 1992. Value bars: An information visualization and navigation tool for multi-attribute listings. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '92)*, Monterey, CA. ACM, 293–294. DOI: https://doi.org/10.1145/142750.142817.

J. P. Chin, V. A. Diehl, and K. L. Norman. 1988. Development of an instrument measuring user satisfaction of the human–computer interface. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '88)*, Washington, DC. ACM, 213–218. DOI: https://doi.org/10.1145/57167.57203.

R. Choudhary and P. Dewan. 1995. A general multi-user undo/redo model. In *Proceedings of the 4th European Conference on Computer-Supported Cooperative Work (ECSCW '95)*, Stockholm, Sweden. Springer, Netherlands, 231–246. DOI: https://doi.org/10.1007/978-94-011-0349-7_15.

C. Christensen. 1968. An example of the manipulation of directed graphs in the AMBIT/G programming language. In M. Klerer and J. Reinfelds (Eds.), *Interactive Systems for Experimental Applied Mathematics.* Academic Press, 423–435.

D. J. Clarke. 2002. MIT grad directs Spielberg in the science of moviemaking. *MIT News*. July 17, 2002. http://news.mit.edu/2002/underkoffler-0717.

E. Clarkson, J. Clawson, K. Lyons, and T. Starner. April 2–7. 2005. An empirical study of typing rates on mini-QWERTY keyboards. In *CHI '05 Extended Abstracts on Human Factors in Computing Systems (CHI EA '05)*, Portland, OR. ACM, 1288–1291. DOI: https://doi.org/10.1145/1056808.1056898.

M. J. Coblenz, A. J. Ko, and B. A. Myers. October 22–23. 2006. JASPER: An eclipse plug-in to facilitate software maintenance tasks. In *Eclipse Technology eXchange (ETX) Workshop at OOPSLA 2006*, Portland, OR. ACM, 65–69. DOI: https://doi.org/10.1145/1188835.1188849.

A. Cockburn and C. Gutwin. June 15. 2009. A predictive model of human performance with scrolling and hierarchical Lists. *Hum. Comput. Interact.* 24, 3, 273–314. DOI: https://doi.org/10.1080/07370020902990402.

A. Cohé, F. Dècle, and M. Hachet. 2011. tBox: A 3D transformation widget designed for touch-screens. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '11)*, Vancouver, BC. ACM, 3005–3008. DOI: https://doi.org/10.1145/1978942.1979387.

M. L. Coleman. 1969. Text editing on a graphic display device using hand-drawn proofreader's symbols. In M. Faiman and J. Nievergelt (Eds.), *Pertinent Concepts in Computer Graphics.* University of Illinois Press, Urbana, IL, 282–290.

S. Coons. 1963. An outline of the requirements for a computer-aided design system. In *AFIPS Spring Joint Computer Conference*. ACM, 299–304. https://doi.org/10.1145/1461551.1461588.

C. Cruz-Neira, D. J. Sandin, T. A. DeFanti, R. V. Kenyon, and J. C. Hart. 1992. The CAVE: Audio visual experience automatic virtual environment. *Commun. ACM* 35, 6, 64–72. DOI: https://doi.org/10.1145/129888.129892.

T. Cubitt. 2020. Undo-Tree.el Version 0.7.4 for Emacs. Free Software Foundation. http://www.emacswiki.org/emacs/UndoTree and http://www.dr-qubit.org/undo-tree/undo-tree.el.

A. Cypher, D. C. Halbert, D. Kurlander, H. Lieberman, D. Maulsby, B. A. Myers, and A. Turransky. 1993. *Watch What I Do: Programming by Demonstration*. MIT Press, Cambridge, MA.

R. I. Damper. December. 1984. Text composition by the physically disabled: A rate prediction model for scanning input. *Appl. Ergon.* 15, 4, 289–296. DOI: https://doi.org/10.1016/0003-6870(84)90201-1.

J. J. Darragh, I. H. Witten, and M. L. James. November. 1990. The Reactive Keyboard: A predictive typing aid. *Computer* 23, 11, 41–49. DOI: https://doi.org/10.1109/2.60879.

M. R. Davis and T. O. Ellis. 1964. The RAND tablet: A man–machine graphical communication device. In *Proceedings of the Fall Joint Computer Conference (AFIPS '64)*. ACM, 325–331. DOI: https://doi.org/10.1145/1464052.1464080. https://www.rand.org/content/dam/rand/pubs/research_memoranda/2005/RM4122.pdf.

B. Demain. January 3. 2012. When the car radio was introduced, people freaked out. *Mental Floss*. https://www.mentalfloss.com/article/29631/when-car-radio-was-introduced-people-freaked-out.

M. Dertouzos. 1997. *What Will Be*. HarperEdge, San Francisco.

P. Dietz and D. Leigh. 2001. DiamondTouch: A multi-user touch technology. In *Proceedings of the 14th Annual ACM Symposium on User Interface Software and Technology (UIST '2001)*, Orlando, Florida. ACM, 219–226. DOI: https://doi.org/10.1145/502348.502389.

P. Dragicevic. 2004. Combining crossing-based and paper-based interaction paradigms for dragging and dropping between overlapping windows. In *Proceedings of the 17th Annual ACM Symposium on User Interface Software and Technology (UIST '2004)*, Santa Fe, NM. ACM, 193–196. DOI: https://doi.org/10.1145/1029632.1029667.

A. Druin. 1988. NOOBIE: The animal design playstation. *SIGCHI Bull.* 20, 1, 45–53. DOI: https://doi.org/10.1145/49103.49106.

K. Dye and C. Graham. 1995. Chapter 6: User knowledge and user-centered interface design in Microsoft applications. In J. Nielsen (Ed.), *Advances in Human–Computer Interaction*, Vol. 5. Ablex, NJ, 159–190.

S. G. Eick. 1994. Data visualization sliders. In *Proceedings of the 7th Annual ACM Symposium on User Interface Software and Technology (UIST '94)*, Marina del Rey, CA. ACM, 119–120. DOI: https://doi.org/10.1145/192426.192472.

S. G. Eick and J. L. Steffen. 1992. Visualizing code profiling line oriented statistics. In *Proceedings of the IEEE Visualization*, Boston, MA. IEEE, 210–217. DOI: https://doi.org/10.1109/VISUAL.1992.235206.

D. Engber. March 9. 2014. Who made that progress bar? *New York Times Sunday Magazine* MM18. https://www.nytimes.com/2014/03/09/magazine/who-made-that-progress-bar.html.

D. Engelbart. 1988. The augmented knowledge workshop. In A. Goldberg (Ed.), *A History of Personal Workstations.* Addison-Wesley, New York, NY, 187–232.

D. Engelbart and W. English. 1968. A research center for augmenting human intellect. Reprinted in *ACM SIGGRAPH Video Review*, 1994. (90 minutes). Vol. 106. http://vimeo.com/1408300.

W. K. English, D. C. Engelbart, and M. L. Berman. March. 1967. Display selection techniques for text manipulation. *IEEE Trans. Hum. Factors Electron*. HFE-8, 1, 5–15. DOI: https://doi.org/10.1109/THFE.1967.232994.

J. Evans. March 17. 2010. I can type 54 WPM on my iPhone. *The Mobile Gadgeteer* at *ZDNet.com.* https://www.zdnet.com/article/i-can-type-54-wpm-on-my-iphone/.

A. Faeth, M. Oren, and C. Harding. 2008. Combining 3-D geovisualization with force feedback driven user interaction. In *Proceedings of the 16th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (GIS '08)*, Irvine, CA. ACM, 25. DOI: https://doi.org/10.1145/1463434.1463466.

B. Feldman. October 31. 2016. Clippy didn't just annoy you—He changed the world. *New York Magazine*. https://nymag.com/vindicated/2016/10/clippy-didnt-just-annoy-you-he-changed-the-world.html.

S. Fertig, E. Freeman, and D. Gelernter. 1996. Lifestreams: An alternative to the desktop metaphor. In *Proceedings of the Conference Companion on Human Factors in Computing Systems: Common Ground*, Vancouver, BC. ACM, 410–411. DOI: https://doi.org/10.1145/257089.257404.

R. D. Fields. December 13. 2010. Why can some blind people process speech far faster than sighted persons? *Sci. Am.* https://www.scientificamerican.com/article/why-can-some-blind-people-process/.

S. Finger, M. Terk, E. Subrahamanian, C. Kasabach, F. Prinz, D. P. Siewiorek, A. Smailagic, J. Stivoric, and L. Weiss. 1996. Rapid design and manufacture of wearable computers. *Commun. ACM* 39, 2, 63–70. DOI: https://doi.org/10.1145/230798.230806.

P. M. Fitts. 1954. The information capacity of the human motor system in controlling amplitude of movement. *J. Exp. Psychol.* 47, 6, 381–391. DOI: https://psycnet.apa.org/doi/10.1037/h0055392. (Reprinted in *J. Exp. Psychol. Gen.* 1992, 121, 3, 262–269. DOI: https://psycnet.apa.org/doi/10.1037/0096-3445.121.3.262.). https://lab.brainonfire.net/drop/Paul%20Fitts%201954.pdf.

G. W. Fitzmaurice. 1993. Situated information spaces and spatially aware palmtop computers. *Commun. ACM* 36, 7, 39–49. DOI: https://doi.org/10.1145/159544.159566.

G. W. Fitzmaurice and W. Buxton. 1997. An empirical evaluation of graspable user interfaces: Towards specialized, space-multiplexed input. In *Proceedings of CHI'97: the ACM SIGCHI Conference on Human Factors in Computing Systems*. ACM, Atlanta, GA, 43–50. DOI: https://doi.org/10.1145/258549.258578.

G. W. Fitzmaurice, H. Ishii, and W. A. S. Buxton. 1995. Bricks: Laying the foundations for graspable user interfaces. In *Proceedings CHI'95: the SIGCHI Conference on Human Factors in Computing Systems*. ACM, Denver, CO, 442–449. DOI: https://doi.org/10.1145/223904.223964.

S. Flynn. September. 2005. DigiBarn virtual visitors comment on the GRiDpad. *DigiBarn*. http://www.digibarn.com/collections/systems/gridpad/index.html.

J. D. Foley and V. L. Wallace. April. 1974. The art of natural graphic man–machine conversation. *Proc. IEEE.* 62, 4, 462–471. DOI: https://doi.org/10.1109/PROC.1974.9450.

J. D. Foley and A. van Dam. 1982. *Fundamentals of Interactive Computer Graphics*. Addison-Wesley, Reading, MA, 664.

J. D. Foley, V. L. Wallace, and P. Chan. November. 1984. The human factors of computer graphics interaction techniques. *IEEE Comput. Graph. Appl.* 4, 11, 13–48. DOI: https://doi.org/10.1109/MCG.1984.6429355.

J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. 1990. *Computer Graphics; Principles and Practice* (2nd. ed.). Addison-Wesley, Reading, MA, 1174.

A. Fowler. August. 1998. A Swing Architecture Overview: The Inside Story on JFC Component Design. Sun Microsystems. https://www.oracle.com/java/technologies/a-swing-architecture.html.

A. Fowler, K. Partridge, C. Chelba, X. Bi, T. Ouyang, and S. Zhai. 2015. Effects of language modeling and its personalization on touchscreen typing performance. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems (CHI'15)*, Seoul, Republic of Korea. ACM, 649–658. DOI: https://doi.org/10.1145/2702123.2702503.

M. R. Frank and J. D. Foley. November. 1993. Model-based user interface design by example and by interview. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology (UIST '93)*, Atlanta, GA. ACM, 129–137. DOI: https://doi.org/10.1145/168642.168655.

B. N. Freeman-Benson. October. 1990. Kaleidoscope: Mixing objects, constraints, and imperative programming. In *Proceedings of the European Conference on Object-Oriented Programming on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA/ECOOP '90)*, Ottawa, ON. ACM, 77–88. DOI: https://doi.org/10.1145/97945.97957.

K. Gajos and D. Weld. 2004. SUPPLE: Automatically generating user interfaces. In *Proceedings of the 9th International Conference on Intelligent User Interfaces (IUI '04)*, Funchal, Portugal. ACM, 93–100. DOI: https://doi.org/10.1145/964442.964461.

E. Gamma, R. Helm, R. Johnson, and J. Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA.

M. Gancarz. June 10–13. 1986. UWM: A user interface for X windows. In *Proceedings of the USENIX Association Summer Technical Conference*, Atlanta, GA. USENIX Association, 429–440.

L. Gannes. August 23. 2013. Nearly a decade later, the autocomplete origin story: Kevin Gibbs and Google Suggest. *All Things D*. http://allthingsd.com/20130823/nearly-a-decade-later-the-autocomplete-origin-story-kevin-gibbs-and-google-suggest/.

GBD—2015 Disease and Injury Incidence and Prevalence Collaborators GBD. 2016. Global, regional, and national incidence, prevalence, and years lived with disability for 310 diseases and injuries, 1990–2015: A systematic analysis for the Global Burden of Disease

Study 2015. *Lancet* 388, 10053, 1545–1602. https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5055577/. DOI: https://doi.org/10.1016%2FS0140-6736(16)31678-6.

D. Gelernter. 1993. *Mirror Worlds: Or the Day Software Puts the Universe in a Shoebox...How It Will Happen and What It Will Mean*. Oxford University Press.

M. Gleicher. November. 1993. A graphics toolkit based on differential constraints. In *Proceedings of the 6th Annual ACM Symposium on User Interface Software and Technology (UIST '93)*, Atlanta, GA. ACM, 109–120. DOI: https://doi.org/10.1145/168642.168653.

E. P. Glinert and B. W. York. 1992. Computers and people with disabilities. *Commun. ACM* 35, 5, 32–35. DOI: https://doi.org/10.1145/129875.129876.

Go Corporation. 1992. *Getting Started with PenPoint, Version 1.0*. https://www.microsoft.com/buxtoncollection/a/pdf/go%20penpoint%20getting%20started.pdf.

A. Goldberg (Ed.). 1988. *A History of Personal Workstations*. Addison-Wesley, New York, NY.

D. Goldberg. 1993. Unistrokes for computerized interpretation of handwriting. USA Patent No. 5596656, Filed Oct. 6, 1993, Issued Jan. 21, 1997.

A. Goldberg and A. Kay (Eds.). March. 1976. *Smalltalk-72 Instruction Manual*. Xerox Palo Alto Research Center Report SSL 76-6. Xerox Corporation.

A. Goldberg and D. Robson. 1983. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, Reading, MA.

D. Goldberg and C. Richardson. April. 1993. Touch-typing with a stylus. In *Proceedings of the Human Factors in Computing Systems (INTERCHI'93)*, Amsterdam, Netherlands. ACM, 80–87. DOI: https://doi.org/10.1145/169059.169093.

I. E. Gonzalez, J. O. Wobbrock, D. H. Chau, A. Faulring, and B. A. Myers. May 28–30. 2007. Eyes on the road, hands on the wheel: Thumb-based interaction techniques for input on steering wheels. *Proceedings of the Graphics Interface (GI' 07)*, Montreal, QC. ACM, 95–102. DOI: https://doi.org/10.1145/1268517.1268535.

M. Gordon, T. Ouyang, and S. Zhai. 2016. WatchWriter: Tap and gesture typing on a smartwatch miniature keyboard with statistical decoding. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, San Jose, CA. ACM, 3817–3821. DOI: http://dx.doi.org/10.1145/2858036.2858242.

J. Gosling. 1986a. *NeWS: A Definitive Approach to Window Systems*. Sun Microsystems Corp., Mountain View, CA.

J. Gosling. 1986b. SunDew—A distributed and extensible window system. In *Proceedings of an Alvey Workshop on Methodology of Window Management*, Abingdon, UK. Springer-Verlag, 47–57. (A slightly different version of this paper with the same title also appeared in the *Proc. 1986 Winter USENIX Tech. Conf.*, Jan. 1986, 98–103). DOI: https://doi.org/10.1007/978-3-642-70919-7_5.

S. Greenberg and C. Fitchett. 2001. Phidgets: Easy development of physical interfaces through physical widgets. In *Proceedings of the 14th Annual ACM Symposium on User Interface Software and Technology (UIST'2001)*, Orlando, FL. ACM, 209–218. DOI: https://doi.org/10.1145/502348.502388.

I. Greif and S. Sarin. 1987. Data sharing in group work. *ACM Trans. Inform. Syst.* 5, 2, 187–211. DOI: https://doi.org/10.1145/27636.27640.

J. T. Griffin, J. A. Holmes, M. Lazaridis, H. A. Little, and H. R. Major. 1998. Hand-held electronic device with a keyboard optimized for use with the thumbs. USA Patent No. 6278442B1, Filed June 29, 1998, Issued Aug. 21, 2001.

T. Grossman and R. Balakrishnan. 2005. The bubble cursor: Enhancing target acquisition by dynamic resizing of the cursor's activation area. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '05)*, Portland, OR. ACM, 281–290. DOI: https://doi.org/10.1145/1054972.1055012.

D. L. Grover, M. T. King, and C. A. Kushler. 1995. Reduced keyboard disambiguating computer. USA Patent No. 5818437, Filed July 26, 1995, Issued Oct. 6, 1998.

J. Grudin. 2017. *From Tool to Partner: The Evolution of Human–Computer Interaction*. Morgan & Claypool.

C. Guglielmo. 2012. Apple wins over jury in Samsung patent dispute, awarded $1.05 billion in damages. *Forbes*. https://www.forbes.com/sites/connieguglielmo/2012/08/24/jury-has-reached-verdict-in-apple-samsung-patent-suit-court-to-announce-it-shortly/#2d5ffe95b711.

Y. Guiard. 1987. Asymmetric division of labor in human skilled bimanual action: The kinematic chain as a model. *J. Motor Behav.* 19, 4, 486–517. DOI: https://doi.org/10.1080/00222895.1987.10735426.

A. Guo, J. Kim, X. A. Chen, T. Yeh, S. E. Hudson, J. Mankoff, and J. P. Bigham. 2017. Facade: Auto-generating tactile interfaces to appliances. In *Proceedings of the Conference on Human Factors in Computing Systems (CHI '2017).* ACM, 5826–5838. DOI: https://doi.org/10.1145/3025453.3025845.

F. Halasz and T. P. Moran. 1982. Analogy considered harmful. In *Proceedings of the 1982 Conference on Human Factors in Computing Systems*, Gaithersburg, MD. ACM, 383–386. DOI: https://doi.org/10.1145/800049.801816.

D. C. Halbert. 1993. SmallStar: Programming by demonstration in the desktop metaphor. In A. Cypher (Ed.), *Watch What I Do: Programming by Demonstration.* MIT Press, Cambridge, MA, 102–123.

K. L. Halsted and J. H. Roberts. 2002. Eclipse help system: An open source user assistance offering. In *Proceedings of the 20th Annual International Conference on Computer Documentation (SIGDOC '02)*, Toronto, ON. ACM, 49–59. DOI: https://doi.org/10.1145/584955.584964.

R. Hamelin. July 13. 2010. Swiftkey keyboard for Android now available in the market as a public beta. Swype begone—It's that good! *Android Police.* 11:08pm PDT, 2010. https://www.androidpolice.com/2010/07/13/swiftkey-keyboard-for-android-now-available-in-the-market-as-a-public-beta-swype-begone-its-that-good/.

H. Hanselmann. May. 2008. Challenges in automotive software engineering. In *Proceedings of the Companion of the 30th International Conference on Software Engineering (ICSE '2008)*, Leipzig, Germany. ACM, 888. DOI: https://doi.org/10.1145/1370175.1370178.

J. P. Hansen, K. Tørning, A. S. Johansen, K. Itoh, and H. Aoki. 2004. Gaze typing compared with input by head and hand. In *Proceedings of the 2004 Symposium on Eye Tracking Research & Applications*, San Antonio, TX. ACM, 131–138. DOI: https://doi.org/10.1145/968363.968389.

D. Harel. May. 1988. On visual formalisms. *Commun. ACM* 31, 5, 514–530. DOI: https://doi.org/10.1145/42411.42414.

A. Harley. June 3. 2018. Visibility of system status (Usability Heuristic #1). *Nielsen Norman Group*. https://www.nngroup.com/articles/visibility-system-status/.

C. Harrison and H. Faste. 2014. Implications of location and touch for on-body projected interfaces. In *Proceedings of the Conference on Designing Interactive Systems (DIS '2014)*, Vancouver, BC. ACM, 543–552. DOI: https://doi.org/10.1145/2598510.2598587.

C. Harrison, B. Amento, S. Kuznetsov, and R. Bell. 2007. Rethinking the progress bar. In *Proceedings of the 20th Annual ACM Symposium on User Interface Software and Technology (UIST '07)*, Newport, RI. ACM, 115–118. DOI: https://doi.org/10.1145/1294211.1294231.

C. Harrison, Z. Yeo, and S. E. Hudson. 2010a. Faster progress bars: Manipulating perceived duration with visual augmentations. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI'2010)*, Atlanta, GA. ACM, 1545–1548. DOI: https://doi.org/10.1145/1753326.1753556.

C. Harrison, D. Tan, and D. Morris. 2010b. Skinput: Appropriating the body as an input surface. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '2010)*, Atlanta, GA. ACM, 453–462. DOI: https://doi.org/10.1145/1753326.1753394.

H. R. Hartson and P. S. Pyla. 2012. *The UX Book: Process and Guidelines for Ensuring a Quality User Experience*. Morgan Kaufmann/Elsevier. http://www.theuxbook.net/.

O. Hashimoto and B. A. Myers. November. 1992. Graphical styles for building user interfaces by demonstration. In *Proceedings of the 5th Annual ACM SIGGRAPH Symposium on User Interface Software and Technology (UIST '92)*, Monterey, CA. ACM, 117–124. DOI: https://doi.org/10.1145/142621.142635.

D. A. Henderson Jr. April. 1986. The Trillium user interface design environment. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (SIGCHI '86)*, Boston, MA. ACM, 221–227. DOI: https://doi.org/10.1145/22627.22375.

D. A. Henderson Jr. and S. K. Card. July. 1986. Rooms: The use of multiple virtual workspaces to reduce space contention in a window-based graphical user interface. *ACM Trans. Graph*. 5, 3, 211–243. DOI: https://doi.org/10.1145/24054.24056.

T. R. Henry, S. E. Hudson, A. K. Yeatts, B. A. Myers, and S. Feiner. 1991. A nose gesture interface device: Extending virtual realities. In *Proceedings of the 4th Annual ACM Symposium on User Interface Software and Technology (UIST '91)*, Hilton Head, SC. ACM, 65–68. DOI: https://doi.org/10.1145/120782.120789.

C. F. Herot, R. Carling, M. Friedell, and D. Kramlich. 1980. A prototype spatial data management system. In *Proceedings of the 7th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '80)*. ACM, 63–70. DOI: https://doi.org/10.1145/800250.807470.

R. D. Hill. July. 1986. Supporting concurrency, communication and synchronization in human–computer interaction—The Sassafras UIMS. *ACM Trans. Graph*. 5, 3, 179–210. DOI: https://doi.org/10.1145/24054.24055.

A. Hills. June. 1999. Wireless Andrew. *IEEE Spectr.* 36, 6, 49–53. DOI: https://doi.org/10.1109/6.769269.

K. Hinckley, J. Pierce, M. Sinclair, and E. Horvitz. 2000. Sensing techniques for mobile interaction. In *Proceedings of the 13th Annual ACM Symposium on User Interface Software and Technology (UIST '2000)*, San Diego, CA. ACM, 91–100. DOI: https://doi.org/10.1145/354401.354417.

D. Hopkins. December. 1991. The design and implementation of pie menus. *Dr. Dobb's J. Special Windows .NET Suppl.* 16, 12, 16–26. http://www.donhopkins.com/home/catalog/piemenus/ddj/piemenus.html.

D. Hopkins. February 11. 2019. Pie menu timeline. *Medium*. https://donhopkins.medium.com/pie-menu-timeline-21bec9b21620.

T. Howard. 2014. Journey mapping: A brief overview. *Commun. Des. Q.* 2, 3, 10–13. https://doi.org/10.1145/2644448.2644451.

S. E. Hudson. April. 1993. A system for efficient and flexible one-way constraint evaluation in C++. Graphics Visualization and Usability Center, College of Computing, Georgia Institute of Technology. http://hdl.handle.net/1853/3618.

S. E. Hudson and S. P. Mohamed. July. 1990. Interactive specification of flexible user interface displays. *ACM Trans. Inf. Syst.* 8, 3, 269–288. DOI: https://doi.org/10.1145/98188.98201.

S. E. Hudson and I. Smith. November. 1996. Ultra-lightweight constraints. In *Proceedings of the 9th Annual ACM SIGGRAPH Symposium on User Interface Software and Technology (UIST '96)*, Seattle, WA. ACM, 147–155. DOI: https://doi.org/10.1145/237091.237112.

S. Hudson, J. Mankoff, and I. Smith. 2005. Extensible input handling in the subArctic toolkit. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '2005).* ACM, 381–390. DOI: https://doi.org/10.1145/1054972.1055025.

J. F. Hughes, A. van Dam, M. McGuire, D. F. Sklar, J. D. Foley, S. K. Feiner, and K. Akeley. 2013. *Computer Graphics: Principles and Practice* (3rd. ed.). Addison-Wesley Professional, Reading, MA, 1264.

T. Igarashi, S. Matsuoka, and H. Tanaka. 1999. Teddy: A sketching interface for 3D freeform design. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '99)*, Los Angeles, CA. ACM, 409–416. DOI: https://doi.org/10.1145/311535.311602.

D. H. H. Ingalls. August. 1981. The Smalltalk graphics kernel. *Byte* 6, 8, 168–194.

L. Introna and D. Wood. 2004. Picturing algorithmic surveillance: The politics of facial recognition systems. *Surveill. Soc.* 2, 2/3, 177–198. DOI: https://doi.org/10.24908/ss.v2i2/3.3373.

W. Isaacson. 2011. *Steve Jobs*. Simon & Schuster.

W. Isaacson. 2015. *The Innovators: How a Group of Hackers, Geniuses, and Geeks Created the Digital Revolution*. Simon & Schuster.

H. Ishii and B. Ullmer. March. 1997. Tangible bits: Towards seamless interfaces between people, bits and atoms. In *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems (CHI '97)*, Atlanta, GA. ACM, 234–241. DOI: https://doi.org/10.1145/258549.258715.

ISO. 2008. *ISO 9241-171:2008: Ergonomics of Human–System Interaction—Part 171: Guidance on Software Accessibility*. Technical Committee: ISO/TC 159/SC 4 Ergonomics of Human–System Interaction. https://www.iso.org/standard/39080.html.

S. Izadi. 2016. The reality of mixed reality. In *Proceedings of the 2016 Symposium on Spatial User Interaction*, Tokyo, Japan. ACM, 1–2. DOI: https://doi.org/10.1145/2983310.2983311.

R. J. K. Jacob. August. 1985. A state transition diagram language for visual programming. *IEEE Comput.* 18, 8, 51–59. DOI: https://doi.org/10.1109/MC.1985.1662976.

M. Jakobsson. 1986. Autocompletion in full text transaction entry: A method for humanized input. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, Boston, MA. ACM, 327–332. DOI: https://doi.org/10.1145/22627.22391.

B. E. John and D. E. Kieras. 1996. The GOMS family of user interface analysis techniques: Comparison and contrast. *ACM Trans. Comput. Hum. Interact.* 3, 4, 320–351. DOI: https://doi.org/10.1145/235833.236054.

T. Johnson. 1963. Sketchpad III: Three dimensional graphical communication with a digital computer. In *AFIPS Spring Joint Computer Conference*. ACM, 347–353. DOI: https://doi.org/10.1145/1461551.1461592.

J. Johnson, T. L. Roberts, W. Verplank, D. C. Smith, C. H. Irby, M. Beard, and K. Mackey. 1989. The Xerox Star: A retrospective. *Computer* 22, 9, 11–26. DOI: https://doi.org/10.1109/2.35211.

K. R. Johnston. 1994. Method of displaying multiple sets of information in the same area of a computer screen. USA Patent No. 5546528, Filed June 23, 1994, Issued Aug. 13, 1996.

J. Joyce. August. 1984. Interview with Bill Joy. *Unix Review*. Archived at: https://web.archive.org/web/20120210184000/http://web.cecs.pdx.edu/~kirkenda/joy84.html.

H. Kacorri, M. Huenerfauth, S. Ebling, K. Patel, K. Menzies, and M. Willard. 2017. Regression analysis of demographic and technology-experience factors influencing acceptance of sign language animation. *ACM Trans. Access. Comput.* 10, 1, 1–33. DOI: https://doi.org/10.1145/3046787.

D. J. Kasik. July. 1982. A user interface management system. In *Proceedings of the 9th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '82)*, Boston, MA. ACM, 99–106. DOI: https://doi.org/10.1145/800064.801268.

A. Kay. 1969. *The Reactive Engine*. Ph.D. thesis. Electrical Engineering and Computer Science, University of Utah.

A. Kay. 1977. Personal dynamic media. *IEEE Comput.* 10, 3, 31–42. DOI: https://doi.org/10.1109/C-M.1977.217672.

S. Keates, P. J. Clarkson, and P. Robinson. 1998. Developing a methodology for the design of accessible interfaces. In *Proceedings of the 4th ERCIM Workshop on User Interfaces for All*, Stockholm. The European Research Consortium for Informatics and Mathematics – ERCIM, 1–15. http://ui4all.ics.forth.gr/UI4ALL-98/keates.pdf.

S. Keates, P. J. Clarkson, L.-A. Harrison, and P. Robinson. 2000. Towards a practical inclusive design approach. In *Proceedings of the ACM Conference on Universal Usability (CUU '00)*. ACM, 45–52. DOI: https://doi.org/10.1145/355460.355471.

C. Kelleher and R. Pausch. 2005. Stencils-based tutorials: Design and evaluation. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '05)*, Portland, OR. ACM, 541–550. DOI: https://doi.org/10.1145/1054972.1055047.

C. Kelleher, R. Pausch, and S. Kiesler. 2007. Storytelling Alice motivates middle school girls to learn computer programming. In *Proceedings of the Conference on Human Factors in Computing Systems (CHI '07)*. ACM, 1455–1464. DOI: https://doi.org/10.1145/1240624.1240844.

W. C. Kim and J. D. Foley. April. 1993. Providing high-level control and expert assistance in the user interface presentation design. In *Proceedings of the INTERACT '93 and CHI '93 Conference on Human Factors in Computing Systems (CHI '93)*, Amsterdam, Netherlands. ACM, 430–437. DOI: https://doi.org/10.1145/169059.169346.

A. J. Ko and B. A. Myers. April 22–27. 2006. Barista: An implementation framework for enabling new tools, interaction techniques and views in code editors. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '06)*, Montreal, QC. ACM, 387–396. DOI: https://doi.org/10.1145/1124772.1124831.

A. J. Ko, B. A. Myers, and H. H. Aung. September 26–29. 2004. Six learning barriers in end-user programming systems. In *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, Rome, Italy. IEEE, 199–206. DOI: https://doi.org/10.1109/VLHCC.2004.47.

A. J. Ko, H. H. Aung, and B. A. Myers. April 2–7. 2005. Design requirements for more flexible structured editors from a study of programmers' text editing. In *Extended Abstracts CHI'05: Human Factors in Computing Systems (CHI EA '05)*, Portland, OR. ACM, 1557–1560. DOI: https://doi.org/10.1145/1056808.1056965.

A. J. Ko, B. A. Myers, M. Coblenz, and H. H. Aung. December. 2006. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Trans. Softw. Eng.* 33, 12, 971–987. DOI: https://doi.org/10.1109/TSE.2006.116.

M. Kobayashi and T. Igarashi. 2003. Considering the direction of cursor movement for efficient traversal of cascading menus. In *Proceedings of the UIST'03: The 16th Annual ACM Symposium on User Interface Software and Technology*, Vancouver, BC. ACM, 91–94. DOI: https://doi.org/10.1145/964696.964706.

J. L. Kodosky, J. J. Truchard, and J. E. MacCrisken. 1986. Graphical system for modelling a process and associated method. USA Patent No. 4901221, Filed April 14, 1986, Issued Feb. 13, 1990.

J. Kodosky, J. MacCrisken, and G. Rymar. October 8–11. 1991. Visual programming using structured data flow. In *Proceedings of the IEEE Workshop on Visual Languages (VL'91)*. IEEE, 34–39. DOI: https://doi.org/10.1109/WVL.1991.238853.

N. Kostanjsek, A. Good, R. H. Madden, T. B. Üstün, S. Chatterji, C. D. Mathers, and A. Officer. June 01. 2013. Counting disability: Global and national estimation. *Disabil. Rehabil.* 35, 13, 1065–1069. DOI: https://doi.org/10.3109/09638288.2012.720354.

L. Koved and B. Shneiderman. 1986. Embedded menus: Selecting items in context. *Commun. ACM* 4, 29, 312–318. DOI: https://doi.org/10.1145/5684.5687.

G. E. Krasner and S. T. Pope. August. 1988. A cookbook for using the model–view–controller user interface paradigm in Smalltalk-80. *J. Object Oriented Program.* 1, 3, 26–49.

P.-O. Kristensson and S. Zhai. 2004. SHARK2: A large vocabulary shorthand writing system for pen-based computers. In *Proceedings of the 17th Annual ACM Symposium on User Interface Software and Technology (UIST '04)*, Santa Fe, NM. ACM, 43–52. DOI: https://doi.org/10.1145/1029632.1029640.

P. O. Kristensson and K. Vertanen. 2012. Performance comparisons of phrase sets and presentation styles for text entry evaluations. In *Proceedings of the 2012 ACM International Conference on Intelligent User Interfaces (IUI '12)*, Lisbon, Portugal. ACM, 29–32. DOI: https://doi.org/10.1145/2166966.2166972.

M. W. Krueger, T. Gionfriddo, and K. Hinrichsen. 1985. VIDEOPLACE—An artificial reality. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '85)*, San Francisco, CA. ACM, 35–40. DOI: https://doi.org/10.1145/317456.317463. See also: https://www.youtube.com/watch?v=d4DUIeXSEpk.

K. Kukich. 1992. Techniques for automatically correcting words in text. *ACM Comput. Surv.* 24, 4, 377–439. DOI: https://doi.org/10.1145/146370.146380.

D. Kurlander and E. A. Bier. August 1–5. 1988. Graphical search and replace. In *Computer Graphics, Proceedings SIGGRAPH'88*, Atlanta, GA. ACM, 113–120. DOI: https://doi.org/10.1145/54852.378495.

D. Kurlander and S. Feiner. October. 1988. Editable graphical histories. In *Proceedings of the 1988 IEEE Workshop on Visual Languages*, Pittsburgh, PA. IEEE, 127–134. DOI: https://doi.org/10.1109/WVL.1988.18020.

G. Kurtenbach and W. Buxton. 1993. The limits of expert performance using hierarchic marking menus. In *Proceedings of the INTERACT '93 and CHI '93 Conference on Human Factors in Computing Systems (CHI '93)*, Amsterdam, Netherlands. ACM, 482–487. DOI: https://doi.org/10.1145/169059.169426.

C. A. Kushler and R. J. Marsden. 2006. System and method for continuous stroke word-based text input. USA Patent No. 7098896, Filed Jan. 16, 2003, Issued Aug. 29, 2006.

B. Lafreniere, T. R. Jonker, S. Santosa, M. Parent, M. Glueck, T. Grossman, H. Benko, and D. Wigdor. 2021. False positives vs. false negatives: The effects of recovery time and cognitive costs on input error preference. In *Proceedings of the 34th Annual ACM Symposium on User Interface Software and Technology (UIST '21)*. ACM, 54–68. DOI: https://doi.org/10.1145/3472749.3474735.

L. Lamport. October 1. 1985. *LATEX–A Document Preparation System* (1st. ed.). Addison–Wesley.

J. A. Landay and B. A. Myers. April. 1993. Extending an existing user interface toolkit to support gesture recognition. In *Proceedings of the INTERACT '93 and CHI '93 Conference Companion on Human Factors in Computing Systems (CHI '93)*, Amsterdam, Netherlands. ACM, 91–92. DOI: https://doi.org/10.1145/259964.260123.

J. Landay and B. A. Myers. May. 1995. Interactive sketching for the early stages of user interface design. In *Proceedings of the SIGCHI'95 Conference on Human Factors in Computing Systems (CHI '95)*, Denver, CO. ACM, 43–50. DOI: https://doi.org/10.1145/223904.223910.

J. A. Landay and B. A. Myers. 1996. Sketching storyboards to illustrate interface behaviors. In *Proceedings of the Conference Companion on Human Factors in Computing Systems: Common Ground (SIGCHI '96)*, Vancouver, BC. ACM, 193–194. DOI: https://doi.org/10.1145/257089.257257.

K. Larson and M. Czerwinski. 1998. Web page design: Implications of memory, structure and scent for information retrieval. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '98)*, Los Angeles, CA. ACM, 25–32. DOI: https://doi.org/10.1145/274644.274649.

J. J. LaViola, E. Kruijff, D. Bowman, I. P. Poupyrev, and R. P. McMahan. 2017. *3D User Interfaces: Theory and Practice* (2nd. ed.). Addison-Wesley Professional.

J. Lazar, J. H. Feng, and H. Hochheiser. 2017. *Research Methods in Human–Computer Interaction* (2nd. ed.). Morgan Kaufmann.

C. Levinthal. 1966. Molecular model-building by computer. *Sci. Am.* 214, 6, 42–52. DOI: https://doi.org/10.1038/scientificamerican0666-42.

S. Levy. 1984. *Hackers: Heroes of the Computer Revolution*. Anchor Press/Doubleday, Garden City, NY.

C. Lewis, P. G. Polson, C. Wharton, and J. Rieman. 1990. Testing a walkthrough methodology for theory-based design of walk-up-and-use interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '90)*, Seattle, WA. ACM, 235–242. DOI: https://doi.org/10.1145/97243.97279.

T. J.-J. Li and B. A. Myers. 2021. A need-finding study for understanding text entry in smartphone app usage. arXiv:2105.10127 [cs.HC] 2016. Reprinted as Appendix A of Toby Jia-Jun Li, *A Multi-Modal Intelligent Agent that Learns from Demonstrations and Natural Language Instructions*, Ph.D. Dissertation. Human–Computer Interaction Institute, Carnegie Mellon University, CMU-HCII-21-102, May 3, 2021. https://arxiv.org/abs/2105.10127.

T. Li, A. Azaria, and B. Myers. May 6–11. 2017. SUGILITE: Creating multimodal smartphone automation by demonstration. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems (CHI '17)*, Denver, CO. ACM, 6038–6049. DOI: https://doi.org/10.1145/3025453.3025483.

T. J.-J. Li, I. Labutov, X. N. Li, X. Zhang, W. Shi, W. Ding, T. M. Mitchell, and B. A. Myers. October 1–4. 2018. A multi-modal interface for creating data descriptions in

programming by demonstration using natural language instructions. In *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC '18)*, Lisbon, Portugal, 104–114.

T. J.-J. Li, M. Radensky, J. Jia, K. Singarajah, T. Mitchell, and B. Myers. October 20–23. 2019. PUMICE: A multi-modal agent that learns concepts and conditionals from natural language and demonstrations. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology (UIST '19)*, New Orleans, LA. ACM, 577–589. DOI: https://doi.org/10.1145/3332165.3347899.

T. J.-J. Li, J. Chen, H. Xia, T. Mitchell, and B. Myers. October. 2020. Multi-modal repairs of conversational breakdowns in task-oriented dialogs. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology (UIST '20)*, Minneapolis, MN. ACM, 1094–1107. (SIGCHI Best of UIST Best Paper Award). DOI: https://doi.org/10.1145/3379337.3415820.

J. C. R. Licklider and R. W. Taylor. April. 1968. The computer as communication device. *Sci. Tech.* 21–31.

H. Lieberman (Ed.). 2001. *Your Wish is My Command.* Morgan Kaufmann, San Francisco.

H. Lieberman, H. Liu, P. Singh, and B. Barry. December 15. 2004. Beating common sense into interactive applications. *AI Magazine* 25, 4, 63. DOI: https://doi.org/10.1609/aimag.v25i4.1785, https://ojs.aaai.org/index.php/aimagazine/article/view/1785.

H. Lieberman, F. Paterno, and V. Wulf (Eds.). 2006. *End-User Development*. Springer, Dordrecht, Netherlands.

J. Lin and J. A. Landay. 2008. Employing patterns and layers for early-stage design and prototyping of cross-device user interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI' 08)*, Florence, Italy. ACM, 1313–1322. DOI: https://doi.org/10.1145/1357054.1357260.

J. Lin, M. W. Newman, J. I. Hong, and J. A. Landay. April 1–6. 2000. DENIM: Finding a tighter fit between tools and practice for web site design. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '00)*, The Hague, Netherlands. ACM, 510–517. DOI: https://doi.org/10.1145/332040.332486.

M. A. Linton, J. M. Vlissides, and P. R. Calder. February. 1989. Composing user interfaces with InterViews. *IEEE Comput.* 22, 2, 8–22. DOI: https://doi.org/10.1109/2.19829.

H. L. Lipman. March 30. 1858. Combination of lead-pencil and eraser. USA Patent No. 19783, Issued Mar. 30, 1858.

M. X. Liu, A. Kuznetsov, Y. Kim, J. C. Chang, A. Kittur, and B. A. Myers. 2022. Wigglite: Low-cost information collection and triage. In *Proceedings of the 35th Annual ACM Symposium on User Interface Software and Technology (UIST '22)*, Bend, OR. ACM, 32. DOI: https://doi.org/10.1145/3526113.3545661.

E. Lockerman, S. Wu, A. Rao, J. Lin, N. Bantoc, and B. Myers. October 11–14. 2017. Smartwatch text entry using five to seven physical keys. In *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC '17)*, Raleigh, NC. IEEE, 291–295. DOI: https://doi.org/10.1109/VLHCC.2017.8103481.

A. C. Long, J. A. Landay, and L. A. Rowe. 2001. “Those look similar!” issues in automating gesture design advice. In *Proceedings of the 2001 Workshop on Perceptive User Interfaces (PUI ’01)*, Orlando, FL. ACM, 1–5. DOI: https://doi.org/10.1145/971478.971510.

H. Loranger. February 2. 2014. Infinite scrolling is not for every website. *Nielsen Norman Group*. https://www.nngroup.com/articles/infinite-scrolling/.

F. Ludolph, Y.-Y. Chow, D. Ingalls, S. Wallace, and K. Doyle. October. 1988. The Fabrik programming environment. In *1988 IEEE Workshop on Visual Languages*, Pittsburgh, PA. IEEE, 222–230. DOI: https://doi.org/10.1109/WVL.1988.18032.

K. Lyons, D. Plaisted, and T. Starner. October 31–November 3. 2004. Expert chording text entry on the Twiddler one-handed keyboard. *Eighth International Symposium on Wearable Computers (ISWC ’04)*. IEEE, 94–101. DOI: https://doi.org/10.1109/ISWC.2004.19.

R. L. Mace, G. J. Hardie, and J. P. Place. 1991. Accessible environments: Toward universal design. In W. E. Preiser, J. C. Vischer, and E. T. White (Eds.), *Design Intervention: Toward a More Humane Architecture.* Van Nostrand Reinhold, New York, 1–44.

I. S. MacKenzie. 2002. KSPC (keystrokes per character) as a characteristic of text entry techniques. In *Proceedings of the 4th International Symposium on Mobile Human–Computer Interaction*. Springer-Verlag, 195–210. DOI: https://doi.org/10.1007/3-540-45756-9_16.

I. S. MacKenzie and S. X. Zhang. May 15–20. 1999. The design and evaluation of a high-performance soft keyboard. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI ’99)*, Pittsburgh, PA. ACM, 25–31. DOI: https://doi.org/10.1145/302979.302983.

I. S. MacKenzie and R. W. Soukoreff. 2002a. A character-level error analysis technique for evaluating text entry methods. In *Proceedings of the Second Nordic Conference on Human–Computer Interaction (NordiCHI ’02)*, Aarhus, Denmark. ACM, 243–246. DOI: https://doi.org/10.1145/572020.572056.

I. S. MacKenzie and R. W. Soukoreff. 2002b. Text entry for mobile computing: Models and methods, theory and practice. *Hum. Comput. Interact.* 17, 2 & 3, 147–198. DOI: https://doi.org/10.1207/S15327051HCI172&3_2.

I. S. MacKenzie and R. W. Soukoreff. 2003. Phrase sets for evaluating text entry techniques. In *CHI ’03 Extended Abstracts on Human Factors in Computing Systems (CHI EA ’03)*, Ft. Lauderdale, FL. ACM, 754–755. DOI: https://doi.org/10.1145/765891.765971.

I. S. MacKenzie, J. Chen, and A. Oniszczak. 2006. Unipad: Single stroke text entry with language-based acceleration. In *Proceedings of the 4th Nordic Conference on Human–Computer Interaction: Changing Roles (NordiCHI ’06)*, Oslo, Norway. ACM, 78–85. DOI: https://doi.org/10.1145/1182475.1182484.

J. Mankoff and G. D. Abowd. November 1–4. 1998. Cirrin: A word-level unistroke keyboard for pen input. In *Proceedings of the 11th Annual ACM Symposium on User Interface Software and Technology (UIST ’98)*, San Francisco, CA. ACM, 213–214. DOI: https://doi.org/10.1145/288392.288611.

J. Mankoff, A. K. Dey, G. Hsieh, J. Kientz, S. Lederer, and M. Ames. 2003. Heuristic evaluation of ambient displays. In *Proceedings of the SIGCHI Conference on Human Factors*

in Computing Systems (CHI '03), Ft. Lauderdale, FL. ACM, 169–176. DOI: https://doi.org/10.1145/642611.642642.

T. Marrin. 1997. Possibilities for the digital baton as a general-purpose gestural interface. In *CHI '97 Extended Abstracts on Human Factors in Computing Systems (CHI EA '97)*, Atlanta, GA. ACM, 311–312. DOI: https://doi.org/10.1145/1120212.1120409.

N. Martelaro and W. Ju. 2017. WoZ way: Enabling real-time remote interaction prototyping & observation in on-road vehicles. In *Companion of the 2017 ACM Conference on Computer Supported Cooperative Work and Social Computing (CSCW '17)*, Portland, OR. ACM, 21–24. DOI: https://doi.org/10.1145/3022198.3023271.

T. H. Massie and J. K. Salisbury. November. 1994. The PHANTOM haptic interface: A device for probing virtual objects. In *Proceedings of the ASME Winter Annual Meeting, Symposium on Haptic Interfaces for Virtual Environment and Teleoperator Systems*, Chicago, IL. ASME, 295–301.

T. Masui. April 18–23. 1998. An efficient text input method for pen-based computers. In *Proceedings of the SIGCHI Conference on Human Factors and Computing Systems (CHI '98)*, Los Angeles, CA. ACM, 328–335. DOI: https://doi.org/10.1145/274644.274690.

J. R. Matheny, C. White, and D. B. Goldsmith. 1992. Balloon help system. USA Patent No. 5434965A, Filed Dec. 23, 1992, Issued Jul. 18, 1995. (An Apple subsidiary).

J. McCormack and P. Asente. October. 1988. An overview of the X toolkit. In *Proceedings of the 1st Annual ACM SIGGRAPH Symposium on User Interface Software and Technology (UIST '88)*, Banff, AB. ACM, 46–55. DOI: https://doi.org/10.1145/62402.62407.

D. S. McCrickard and R. Catrambone. 1999. Beyond the scrollbar: An evolution and evaluation of alternative navigation techniques. In *Proceedings of the 1999 IEEE Symposium on Visual Languages*. IEEE, 270–277. DOI: https://doi.org/10.1109/VL.1999.795913.

R. G. McDaniel and B. A. Myers. January. 1998. Building applications using only demonstration. In *Proceedings of the 3rd International Conference on Intelligent User Interfaces (IUI '98)*, San Francisco, CA. ACM, 109–116. DOI: https://doi.org/10.1145/268389.268409.

J. E. McDonald, J. D. Stone, and L. S. Liebelt. October 1. 1983. Searching for items in menus: The effects of organization and type of target. *Proc. Hum. Factors Ergon. Soc. Annu. Meet.* 27, 9, 834–837. DOI: https://doi.org/10.1177/154193128302700919.

P. Mehta, W. Kaye, J. Raymond, R. Wu, T. Larson, R. Punjani, D. Heller, J. Cohen, T. Peters, O. Muravov, and K. Horton. 2018. Prevalence of amyotrophic lateral sclerosis—United States, 2014. *MMWR Morb. Mortal. Wkly. Rep.* 67, 7, 216–218. DOI: https://doi.org/10.15585/mmwr.mm6707a3, https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5858037/.

A. Meschtscherjakov, M. Tscheligi, D. Szostak, S. Krome, B. Pfleging, R. Ratan, I. Politis, S. Baltodano, D. Miller, and W. Ju. 2016. HCI and autonomous vehicles: Contextual experience informs design. In *Proceedings of the 2016 CHI Conference Extended Abstracts on Human Factors in Computing Systems (CHI EA '16)*, San Jose, CA. ACM, 3542–3549. DOI: https://doi.org/10.1145/2851581.2856489.

R. Meyer. June 23. 2015. Even early focus groups hated Clippy. *Atlantic*. https://www.theatlantic.com/technology/archive/2015/06/clippy-the-microsoft-office-assistant-is-the-patriarchys-fault/396653/.

L. A. Miller and J. C. Thomas. September. 1977. Behavioral issues in the use of interactive systems. *Int. J. Man Mach. Stud.* 9, 5, 509–536. DOI: https://doi.org/10.1016/S0020-7373(77)80002-3.

R. C. Miller and B. Myers. November 7–10. 1999. Synchronizing clipboards of multiple computers. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology (UIST '99)*, Asheville, NC. ACM, 65–66. DOI: https://doi.org/10.1145/320719.322584.

T. B. Moeslund and E. Granum. March 01. 2001. A survey of computer vision-based human motion capture. *Comput. Vis. Image Underst.* 81, 3, 231–268. DOI: https://doi.org/10.1006/cviu.2000.0897.

B. Moggridge. 2007. *Designing Interactions*. MIT Press, Cambridge, MA.

A. Mogilevsky. 1995. Method for background spell checking a word processing document. USA Patent No. 5649222, Filed May 8, 1995, Issued Jul. 15, 1997.

M. Mooty, A. Faulring, J. Stylos, and B. A. Myers. September 21–25. 2010. Calcite: Completing code completion for constructors using crowds. In *2010 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC '10)*, Leganés-Madrid, Spain. IEEE, 15–22. DOI: https://doi.org/10.1109/VLHCC.2010.12.

T. P. Moran and P. Dourish. 2001. Special issue on context-aware computing. *HCI J.* 16, 2–4, 87–419. DOI: https://doi.org/10.1207/S15327051HCI16234_01.

J. H. Morris, M. Satyanarayanan, J. H. Howard, M. H. Conner, D. S. H. Rosenthal, and F. D. Smith. March. 1986. Andrew: A distributed personal computing environment. *Commun. ACM* 29, 3, 184–201. DOI: https://doi.org/10.1145/5666.5671.

F. F. Mueller, M. R. Gibbs, and F. Vetere. 2008. Taxonomy of exertion games. In *Proceedings of the 20th Australasian Conference on Computer–Human Interaction: Designing for Habitus and Habitat*, Cairns, Australia. ACM, 263–266. DOI: https://doi.org/10.1145/1517744.1517772.

S. Musil. September 10. 2019. Apple ditches 3D Touch for Haptic Touch in iPhone 11 and 11 Pro. *c|net*. https://www.cnet.com/news/apple-ditches-3d-touch-for-haptic-touch-in-iphone-11-and-11-pro/.

B. A. Myers. June. 1980. *Displaying Data Structures for Interactive Debugging*. MIT master's thesis. Xerox Palo Alto Research Center, Technical Report CSL-80-7. http://www.cs.cmu.edu/~bam/MyersCSL-80-7.pdf.

B. A. Myers. July. 1983. Incense: A system for displaying data structures. In *Proceedings of the Computer Graphics (SIGGRAPH '83)*, Detroit, MI. ACM, 115–125. http://www.cs.cmu.edu/~bam/papers/incense-siggraph83.pdf.

B. A. Myers. December. 1984. The user interface for sapphire. *IEEE Comput. Graph. Appl.* 4, 12, 13–23. DOI: https://doi.org/10.1109/MCG.1984.6429376.

B. A. Myers. April 14–18. 1985a. Percent-done progress indicators in practice and experiments. (Videotape). In *SIGCHI '85*, San Francisco, CA. *SIGGRAPH Video Review*, Issue 19, no. 6. https://youtu.be/gnzaS18XuJ4 or http://www.cs.cmu.edu/~bam/papers/PercentDoneProgressIndicators1985.mp4.

B. A. Myers. April. 1985b. The importance of percent-done progress indicators for computer-human interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '85)*, San Francisco, CA. ACM, 11–17. DOI: https://doi.org/10.1145/1165385.317459.

B. A. Myers. April. 1986a. Visual programming, programming by example, and program visualization: A taxonomy. In *Proceedings of the SIGCHI Human Factors in Computing Systems (CHI '86)*, Boston, MA. ACM, 59–66. DOI: https://doi.org/10.1145/22339.22349.

B. A. Myers. September. 1986b. A complete and efficient implementation of covered windows. *IEEE Comput.* 19, 9, 57–67. DOI: https://doi.org/10.1109/MC.1986.1663332.

B. A. Myers. April. 1987a. Creating dynamic interaction techniques by demonstration. In *Proceedings of the SIGCHI/GI Conference on Human Factors in Computing Systems and Graphics Interface (CHI '87)*, Toronto, ON. ACM, 271–278. DOI: https://doi.org/10.1145/29933.275642.

B. A. Myers. September. 1987b. Creating interaction techniques by demonstration. *IEEE Comput. Graph. Appl.* 7, 9, 51–60. DOI: https://doi.org/10.1109/MCG.1987.277079.

B. A. Myers. September. 1988. A taxonomy of window manager user interfaces. *IEEE Comput. Graph. Appl.* 8, 5, 65–84. DOI: https://doi.org/10.1109/38.7762.

B. A. Myers. April. 1989. Encapsulating interactive behaviors. In *Proceedings of the SIGCHI'89 Conference on Human Factors in Computing Systems*, Austin, TX. ACM, 319–324.

B. A. Myers. July. 1990a. A new model for handling input. *ACM Trans. Inf. Syst.* 8, 3, 289–320. DOI: https://doi.org/10.1145/98188.98204.

B. A. Myers. 1990b. All the Widgets. *SIGGRAPH Video Review*, Vol. 57. https://youtu.be/9qtd8Hc90Hw or https://vimeo.com/61556918.

B. A. Myers. April. 1990c. Creating user interfaces using programming-by-example, visual programming, and constraints. *ACM Trans. Program. Lang. Syst.* 12, 2, 143–177. DOI: https://doi.org/10.1145/78942.78943.

B. A. Myers. November. 1991. Separating application code from toolkits: Eliminating the spaghetti of call-backs. In *Proceedings of the 4th Annual ACM SIGGRAPH Symposium on User Interface Software and Technology (UIST '91)*, Hilton Head, SC. ACM, 211–220. DOI: https://doi.org/10.1145/120782.120805.

B. A. Myers (Ed.). May 3–7. 1992a. CHI'92 Special Video Program: Future Scenarios. *SIGGRAPH Video Review*, Issues 78–79. Monterrey, CA.

B. A. Myers (Ed.). 1992b. *Languages for Developing User Interfaces*. Jones and Bartlett, Boston, MA.

B. A. Myers. January. 1994. Challenges of HCI design and implementation. *Interactions* 1, 1, 73–83. DOI: https://doi.org/10.1145/174800.174808.

B. A. Myers. March. 1995. User interface software tools. *ACM Trans. Comput. Hum. Interact.* 2, 1, 64–103. DOI: https://doi.org/10.1145/200968.200971.

B. A. Myers. March. 1998a. A brief history of human–computer interaction technology. *Interactions* 5, 2, 44–54. DOI: https://doi.org/10.1145/274430.274436.

B. A. Myers. April. 1998b. Scripting graphical applications by demonstration. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (SIGCHI '98)*, Los Angeles, CA. ACM, 534–541. DOI: https://doi.org/10.1145/274644.274716.

B. A. Myers. August. 1998c. *The Case for an Open Data Model*. School of Computer Science Technical Report, CMU-CS-98-153. Carnegie Mellon University. http://reports-archive.adm.cs.cmu.edu/anon/1998/CMU-CS-98-153.pdf.

B. A. Myers. November. 2001. Using hand-held devices and PCs together. *Commun. ACM* 44, 11, 34–41. DOI: https://doi.org/10.1145/384150.384159, http://www.cs.cmu.edu/~pebbles/papers/pebblescacm.pdf.

B. A. Myers and W. Buxton. August. 1986. Creating highly interactive and graphical user interfaces by demonstration. In *Proceedings of the SIGGRAPH'86: Computer Graphics*, Dallas, TX. ACM, 249–258. DOI: https://doi.org/10.1145/1165387.275642.

B. A. Myers and D. Kosbie. April 14–18. 1996. Reusable hierarchical command objects. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '96)*, Vancouver, BC. ACM, 260–267. DOI: https://doi.org/10.1145/238386.238526.

B. A. Myers and J. Stylos. July. 2016. Improving API usability. *Commun. ACM* 59, 6, 62–69. DOI: https://doi.org/10.1145/2896587.

B. A. Myers, R. Chandhok, and A. Sareen. October 10–12. 1988. Automatic data visualization for novice Pascal programmers. In *IEEE Workshop on Visual Languages*, Pittsburgh, PA. IEEE, 192–198. DOI: https://doi.org/10.1109/WVL.1988.18028.

B. A. Myers, B. T. V. Zanden, and R. B. Dannenberg. November. 1989. Creating graphical interactive application objects by demonstration. In *Proceedings of the 2nd Annual ACM SIGGRAPH Symposium on User Interface Software and Technology (UIST '89)*, Williamsburg, VA. ACM, 95–104. DOI: https://doi.org/10.1145/73660.73672.

B. A. Myers, D. A. Giuse, R. B. Dannenberg, B. V. Zanden, D. S. Kosbie, E. Pervin, A. Mickish, and P. Marchal. November. 1990. Garnet: Comprehensive support for graphical, highly-interactive user interfaces. *IEEE Comput.* 23, 11, 71–85. DOI: https://doi.org/10.1109/2.60882.

B. A. Myers, D. C. Smith, and B. Horn. 1992. Report of the "End-User Programming" working group. In *Languages for Developing User Interfaces*. Jones and Bartlett, Boston, MA, 343–366.

B. A. Myers, R. G. McDaniel, and D. S. Kosbie. April. 1993. Marquise: Creating complete user interfaces by demonstration. In *Proceedings of the INTERACT '93 and CHI '93 Conference on Human Factors in Computing Systems (CHI '93)*, Amsterdam, Netherlands. ACM, 293–300. DOI: https://doi.org/10.1145/169059.169225.

B. A. Myers, R. G. McDaniel, R. C. Miller, A. Ferrency, A. Faulring, B. D. Kyle, A. Mickish, A. Klimovitski, and P. Doane. June. 1997a. The Amulet environment: New models for

effective user interface software development. *IEEE Trans. Softw. Eng.* 23, 6, 347–365. DOI: https://doi.org/10.1109/32.601073.

B. A. Myers, A. Ferrency, R. McDaniel, R. C. Miller, P. Doane, A. Mickish, and A. Klimovitski. March. 1997b. *The Amulet V3.0 Reference Manual*. CMU-CS-95-166-R2. Carnegie Mellon University Computer Science Department. http://www.cs.cmu.edu/afs/cs/project/amulet/amulet3/manual/Amulet_ManualTOC.doc.html.

B. A. Myers, H. Stiel, and R. Gargiulo. November 14–18. 1998. Collaboration using multiple PDAs connected to a PC. In *Proceedings of the 1998 ACM Conference on Computer-Supported Cooperative Work (CSCW '98)*, Seattle, WA. ACM, 285–294. DOI: https://doi.org/10.1145/289444.289503, http://www.cs.cmu.edu/~pebbles.

B. A. Myers, S. E. Hudson, and R. Pausch. March. 2000a. Past, present and future of user interface software tools. *ACM Trans. Comput. Hum. Interact.* 7, 1, 3–28. DOI: https://doi.org/10.1145/344949.344959.

B. A. Myers, K. P. Lie, and B.-C. Yang. April 1–6. 2000b. Two-handed input using a PDA and a mouse. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '00)*, The Hague, Netherlands. ACM, 41–48. DOI: https://doi.org/10.1145/332040.332405, http://www.cs.cmu.edu/~pebbles/papers/p41-myersCHI2000final.pdf.

B. A. Myers, R. McDaniel, and D. Wolber. March. 2000c. Intelligence in demonstrational interfaces. *Commun. ACM* 43, 3, 82–89. DOI: https://doi.org/10.1145/330534.330545.

B. A. Myers, R. Bhatnagar, J. Nichols, C. H. Peck, D. Kong, R. Miller, and A. C. Long. April 20–25. 2002a. Interacting at a distance: Measuring the performance of laser pointers and other devices. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (ACM CHI '02)*, Minn, MN. ACM, 33–40. DOI: https://doi.org/10.1145/503376.503383.

B. A. Myers, J. O. Wobbrock, S. Yang, B. Yeung, J. Nichols, and R. Miller. 2002b. Using handhelds to help people with motor impairments. In *Proceedings of the 5th International ACM SIGCAPH Conference on Assistive Technologies (ASSETS '02)*, Scotland. ACM, 89–96. DOI: https://doi.org/10.1145/638249.638266.

B. A. Myers, D. A. Weitzman, A. J. Ko, and D. H. Chau. April 22–27. 2006. Answering why and why not questions in user interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '06)*, Montreal, QC. ACM, 397–406. DOI: https://doi.org/10.1145/1124772.1124832.

B. A. Myers, A. Lai, T. M. Le, Y. S. Yoon, A. Faulring, and J. Brandt. April 18–23. 2015. Selective undo support for painting applications. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, Seoul, Korea. ACM, 4227–4236. DOI: https://doi.org/10.1145/2702123.2702543.

B. A. Myers, A. J. Ko, T. D. LaToza, and Y. S. Yoon. 2016. Programmers are users too: Human-centered methods for improving programming tools. *IEEE Comput.* 49, 7, 44–52. DOI: https://doi.org/10.1109/MC.2016.200.

B. A. Nardi, J. R. Miller, and D. J. Wright. 1998. Collaborative, programmable intelligent agents. *Commun. ACM* 41, 3, 96–104. DOI: https://doi.org/10.1145/272287.272331.

C. Nass and C. Yen. 2010. *The Man Who Lied to His Laptop: What We Can Learn About Ourselves from Our Machines*. Current, Penguin Group.

C. Nass, J. Steuer, and E. R. Tauber. 1994. Computers are social actors. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '94)*, Boston, MA. ACM, 72–78. DOI: https://doi.org/10.1145/259963.260288.

R. Neches, J. Foley, P. Szekely, P. Sukaviriya, P. Luo, S. Kovacevic, and S. Hudson. January. 1993. Knowledgeable development environments using shared design models. In *Proceedings of the 1993 International Workshop on Intelligent User Interfaces*, Orlando, FL. ACM, 63–70. DOI: https://doi.org/10.1145/169891.169901.

T. Nelson. 1965. Complex information processing: A file structure for the complex, the changing, and the indeterminate. In *Proceedings of the ACM National Conference*. ACM, 84–100. DOI: https://doi.org/10.1145/800197.806036.

New York Times. July 23. 1916. Chinaman invents Chinese typewriter using 4,000 characters. https://timesmachine.nytimes.com/timesmachine/1916/07/23/104237769.pdf.

New York Times. December 30. 1922. New world record for shorthand speed. https://timesmachine.nytimes.com/timesmachine/1922/12/30/102911691.pdf.

W. M. Newman. 1968a. A graphical technique for numerical input. *Comput. J.* 11, 1, 63–64. DOI: https://doi.org/10.1093/comjnl/11.1.63.

W. M. Newman. April 30–May 2. 1968b. A system for interactive graphical programming. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference (AFIPS '68)*. ACM, 47–54. DOI: https://doi.org/10.1145/1468075.1468083.

W. M. Newman and R. F. Sproull. 1973. *Principles of Interactive Computer Graphics*. McGraw-Hill, New York.

W. M. Newman and R. F. Sproull. 1979. *Principles of Interactive Computer Graphics* (2nd. ed.). McGraw-Hill, New York.

J. Nichols and B. A. Myers. July–September. 2006. Controlling home and office appliances with smartphones. *IEEE Pervasive Comput.* 5, 3, 60–67. DOI: https://doi.org/10.1109/MPRV.2006.48.

J. Nichols, B. A. Myers, M. Higgins, J. Hughes, T. K. Harris, R. Rosenfeld, and M. Pignol. October. 2002. Generating remote control interfaces for complex appliances. In *Proceedings of the 15th Annual ACM Symposium on User Interface Software and Technology (UIST '02)*, Paris, France. ACM, 161–170. DOI: https://doi.org/10.1145/571985.572008.

J. Nichols, B. A. Myers, and B. Rothrock. April 22–27. 2006a. UNIFORM: Automatically generating consistent remote control user interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '06)*, Montreal, QC. ACM, 611–620. DOI: https://doi.org/10.1145/1124772.1124865.

J. Nichols, B. Rothrock, D. H. Chau, and B. A. Myers. 2006b. Huddle: Automatically generating interfaces for systems of multiple connected appliances. In *Proceedings of the 19th Annual ACM Symposium on User Interface Software and Technology (UIST '06)*, Montreux, Switzerland. ACM, 279–288. DOI: https://doi.org/10.1145/1166253.1166298.

J. Nichols, D. H. Chau, and B. A. Myers. 2007. Demonstrating the viability of automatically generated user interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (SIGCHI '07)*, San Jose, CA. ACM, 1283–1292. DOI: https://doi.org/10.1145/1240624.1240819.

J. Nielsen. May. 1990. Big paybacks from 'discount' usability engineering. *IEEE Softw.* 7, 3, 107–108.

J. Nielsen. 1993. *Usability Engineering*. Academic Press, Boston.

J. Nielsen. 1994. Enhancing the explanatory power of usability heuristics. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '94)*, Boston, MA. ACM, 52–158. DOI: https://doi.org/10.1145/191666.191729. See also: https://www.nngroup.com/articles/ten-usability-heuristics/.

J. Nielsen. 1995. *Multimedia and Hypertext: The Internet and Beyond*. Academic Press Professional, Boston, 480 pp.

J. Nielsen. 1996. Tooltips on webpages. USA Patent No. 5937417, Filed May 7, 1996, Issued Aug. 10, 1999.

J. Nielsen. December 10. 2000. Alertbox: WAP mobile phones field study findings. *Nielsen Norman Group*, Fremont, CA. https://www.nngroup.com/articles/wap-mobile-phones-field-study-findings/.

J. Nielsen and R. Molich. April 1–5. 1990. Heuristic evaluation of user interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '90)*, Seattle, WA. ACM, 249–256. DOI: https://doi.org/10.1145/97243.97281, https://www.nngroup.com/articles/ten-usability-heuristics/.

D. A. Norman. 1988. *The Design of Everyday Things*. Doubleday, New York.

K. L. Norman. 1991. *The Psychology of Menu Selection: Designing Cognitive Control at the Human/Computer Interface*. Greenwood Publishing Group, 352 pp. https://dl.acm.org/doi/book/10.5555/574380.

D. Norman. 1998. *The Invisible Computer: Why Good Products Can Fail, the Personal Computer Is So Complex, and Information Appliances Are the Solution*. MIT Press, Cambridge, MA.

D. Norman and B. Tognazzini. November 10. 2015. How Apple is giving design a bad name. *Co-Design*. https://www.fastcompany.com/3053406/how-apple-is-giving-design-a-bad-name.

NSCISC. February. 2012. National Spinal Cord Injury Statistical Center. University of Alabama, Birmingham, AL. https://www.nscisc.uab.edu/PublicDocuments/fact_figures_docs/Facts%202012%20Feb%20Final.pdf.

D. R. Olsen Jr. April. 1989. A programming language basis for user interface. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (SIGCHI '89)*, Austin, TX. ACM, 171–176. DOI: https://doi.org/10.1145/67449.67485.

D. R. Olsen Jr. 1992. *User Interface Management Systems: Models and Algorithms*. Morgan Kaufmann, San Mateo, CA.

C. Omar, Y. S. Yoon, T. D. LaToza, and B. A. Myers. June 2–9. 2012. Active code completion. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*, Zurich, Switzerland. IEEE, 859–869. DOI: https://doi.org/10.1109/ICSE.2012.6227133.

S. Oney, B. A. Myers, and J. Brandt. October 7–10. 2012. ConstraintJS: Programming interactive behaviors for the Web by integrating constraints and states. In *Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology (UIST '12)*, Cambridge, MA. ACM, 229–238. DOI: https://doi.org/10.1145/2380116.2380146.

S. Oney, C. Harrison, A. Ogan, and J. Wiese. 2013. ZoomBoard: A diminutive QWERTY soft keyboard using iterative zooming for ultra-small devices. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '13)*, Paris, France. ACM, 2799–2802. DOI: https://doi.org/10.1145/2470654.2481387.

S. Oney, B. A. Myers, and J. Brandt. October 5–8. 2014. InterState: A language and environment for expressing interface behavior. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology (UIST '14)*, Honolulu, HI. ACM, 263–272. DOI: https://doi.org/10.1145/2642918.2647358.

S. Oney, R. Krosnick, J. Brandt, and B. Myers. 2019. Implementing multi-touch gestures with touch groups and cross events. In *Proceedings of the Conference on Human Factors in Computing Systems (CHI '19)*, Glasgow, Scotland, UK. ACM, 1–12. DOI: https://doi.org/10.1145/3290605.3300585.

B. Ording. December 23. 2008. List scrolling and document translation, scaling, and rotation on a touch-screen display. USA Patent No. 7469381, Filed Dec. 14, 2007, issued Dec. 23, 2008.

M. Oskoui, F. Coutinho, J. Dykeman, N. Jetté, and T. Pringsheim. 2013. An update on the prevalence of cerebral palsy: A systematic review and meta-analysis. *Dev. Med. Child Neurol.* 55, 6, 509–519. DOI: https://doi.org/10.1111/dmcn.12080.

J. K. Ousterhout. 1994. *Tcl and the Tk Toolkit*. Addison-Wesley.

S. Oviatt. May. 1999. Mutual disambiguation of recognition errors in a multimodel architecture. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '99)*. ACM, 576–583. DOI: https://doi.org/10.1145/302979.303163.

S. Oviatt. September. 2000. Taming recognition errors with a multimodal interface. *Commun. ACM* 43, 9, 45–51. DOI: https://doi.org/10.1145/348941.348979.

S. Oviatt. 2012. Chapter 18: Multimodal interfaces. In J. A. Jacko (Ed.), *Human Computer Interaction Handbook: Fundamentals, Evolving Technologies, and Emerging Applications*, (3rd. ed). CRC Press.

A. J. Palay, W. J. Hansen, M. Kazar, M. Sherman, M. Wadlow, T. Neuendorffer, Z. Stern, M. Bader, and T. Peters. February. 1988. The Andrew toolkit—An overview. In *Proceedings of the Winter USENIX Technical Conference*, Dallas, TX. USENIX Association, 9–21.

S. Papert. 1980. *Mindstorms: Children, Computers, and Powerful Ideas*. ACM Classic Books, Basic Books, New York, 230 pp. https://dl.acm.org/doi/book/10.5555/1095592.

R. Pausch. April. 1991. Virtual reality on five dollars a day. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '91)*. ACM, 265–270. DOI: https://doi.org/10.1145/108844.108913.

R. Pausch, M. Conway, and R. DeLine. October. 1992. Lesson learned from SUIT, the simple user interface toolkit. *ACM Trans. Inf. Syst.* 10, 4, 320–344. DOI: https://doi.org/10.1145/146486.146489.

R. Pausch, T. Burnette, A. C. Capehart, M. Conway, D. Cosgrove, R. DeLine, J. Durbin, R. Gossweiler, S. Koga, and J. White. May. 1995. Alice: A rapid prototyping system for virtual reality. *IEEE Comput. Graph. Appl.* 15, 3, 8–11. DOI: https://doi.org/10.1109/38.376600.

E. Peck, K. Chauncey, A. Girouard, R. Gulotta, F. Lalooses, E. T. Solovey, D. Weaver, and R. Jacob. 2010. From brains to bytes. *XRDS* 16, 4, 42–47. DOI: https://doi.org/10.1145/1764848.17664858.

K. Perlin. November 1–4. 1998. Quikwriting: Continuous stylus-based text entry. In *Proceedings of the 11th Annual ACM Symposium on User Interface Software and Technology (UIST '98)*, San Francisco, CA. ACM, 215–216. DOI: https://doi.org/10.1145/288392.288613.

K. Perlin and D. Fox. 1993. Pad: An alternative approach to the computer interface. In *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '93)*, Anaheim, CA. ACM, 57–64. DOI: https://doi.org/10.1145/166117.166125.

G. R. Pfaff (Ed.). 1985. *User Interface Management Systems*. Springer-Verlag, Berlin.

K. A. Pier. 1983. A retrospective on the Dorado, a high-performance personal computer. In *Proceedings of the 10th Annual International Symposium on Computer Architecture*, Stockholm, Sweden. ACM, 252–269. DOI: https://doi.org/10.1145/800046.801663.

K. Pier, E. Bier, and M. Stone. 1988. An introduction to Gargoyle: An interactive illustration tool. In *Proceedings of the International Conference on Electronic Publishing on Document Manipulation and Typography*, Nice, France. Cambridge University Press, 223–238.

J. S. Pierce, A. S. Forsberg, M. J. Conway, S. Hong, R. C. Zeleznik, and M. R. Mine. 1997. Image plane interaction techniques in 3D immersive environments. In *Proceedings of the 1997 Symposium on Interactive 3D Graphics (I3D '97)*, Providence, RI. ACM, 39–43. DOI: https://doi.org/10.1145/253284.253303.

R. Pike. 1983. Graphics in overlapping bitmap layers. *ACM Trans. Graph.* 2, 2, 135–160. DOI: https://doi.org/10.1145/357318.357322. Also appears in *ACM SIGGRAPH Comput. Graph*. 17, 3, July 1983, 331–355. DOI: https://doi.org/10.1145/964967.801165.

C. Plaisant. August. 1993. Facilitating data exploration: Dynamic queries on a health statistics map. In *Proceedings of the Annual Meeting of the American Statistical Association of the Government Statistics Section (ASA '93)*, Alexandria, VA, 18–23. Also Tech Report: HCIL-93-21. https://hcil.umd.edu/pub-perm-link/?number=93-21.

C. Plaisant and D. Wallace. 1992. Touchscreen toggle design. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '92)*, Monterey, CA. ACM, 667–668. DOI: https://doi.org/10.1145/142750.143079.

O. Polacek, T. Grill, and M. Tscheligi. 2013. NoseTapping: What else can you do with your nose? In *Proceedings of the 12th International Conference on Mobile and Ubiquitous Multimedia (MUM '13)*, Luleå, Sweden. ACM, 1–9. DOI: https://doi.org/10.1145/2541831.2541867.

A. Polaine, L. Løvlie, and B. Reason. 2013. *Service Design: From Insight to Implementation*. Rosenfeld Media.

Polhemus. 2020. *History*. https://polhemus.com/company/history/.

L. Press. 1993. Before the Altair: The history of personal computing. *Commun. ACM* 36, 9, 27–33. DOI: https://doi.org/10.1145/162685.162697.

P. Quinn, S. C. Lee, M. Barnhart, and S. Zhai. 2019. Active Edge: Designing squeeze gestures for the Google Pixel 2. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems (CHI '19)*, Glasgow, Scotland. ACM, 1–13. DOI: https://doi.org/10.1145/3290605.3300504.

M. Rahman, S. Gustafson, P. Irani, and S. Subramanian. 2009. Tilt techniques: Investigating the dexterity of wrist-based input. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '09)*, Boston, MA. ACM, 1943–1952. DOI: https://doi.org/10.1145/1518701.1518997.

D. R. Reddy. 1990. Speech recognition by machine: A review. In A. Waibel and K.-F. Lee (Eds.), *Readings in Speech Recognition.* Morgan Kaufmann, San Mateo, CA, 8–38.

S. Reich. January 30. 2017. How to master the Scribble feature on Apple Watch. *iDB*. https://www.idownloadblog.com/2017/01/30/master-scribble-apple-watch/.

C. Reilly and S. Qi. 2011. *Snapshot of Deaf and Hard of Hearing People, Postsecondary Attendance and Unemployment*. Gallaudet Research Institute. https://www.gallaudet.edu/office-of-international-affairs/demographics/deaf-employment-reports.

J. W. Reiser. 1982. What ever happened to Channel 1? Based on a March 1982 issue of *Radio Electronics Magazine*. Edited and expanded by J. W. Reiser, FCC International Bureau, Rev. 8-4-2000. http://www.tech-notes.tv/History&Trivia/Channel%20One/Channel_1.htm.

J. Rekimoto. 1996. Tilting operations for small screen interfaces. In *Proceedings of the 9th Annual ACM Symposium on User Interface Software and Technology (UIST '96)*, Seattle, WA. ACM, 167–168. DOI: https://doi.org/10.1145/237091.237115.

J. Rekimoto. October 14–17. 1997. Pick-and-drop: A direct manipulation technique for multiple computer environments. In *Proceedings of the 10th Annual ACM Symposium on User Interface Software and Technology (UIST '97)*, Banff, AB. ACM, 31–39. DOI: https://doi.org/10.1145/263407.263505.

J. Rekimoto. 1999. Time-machine computing: A time-centric approach for the information environment. In *Proceedings of the 12th Annual ACM Symposium on User Interface Software and Technology*, Asheville, NC. ACM, 45–54. DOI: https://doi.org/10.1145/320719.322582.

J. Rekimoto and M. Saitoh. May. 1999. Augmented surfaces: A spatially continuous work space for hybrid computing environments. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '99)*, Pittsburgh, PA. ACM, 378–385. DOI: https://doi.org/10.1145/302979.303113.

M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, and Y. Kafai. 2009. Scratch: Programming for all. *Commun. ACM* 52, 11, 60–67. DOI: https://doi.org/10.1145/1592761.1592779, http://scratch.mit.edu/.

M. Ressel and R. Gunzenhäuser. 1999. Reducing the problems of group undo. In *Proceedings of the 1999 ACM International Conference on Supporting Group Work (GROUP '99)*, Phoenix, AZ. ACM, 131–139. DOI: https://doi.org/10.1145/320297.320312.

J. Rhyne. 1987. Dialogue management for gestural interfaces. *ACM SIGGRAPH Comput. Graph.* 21, 2, 137–142. DOI: https://doi.org/10.1145/24919.24933.

A. Riener. 2014. Who cares about trust, grade of traveling & quality of user experience in a world of autonomous cars? In *Adjunct Proceedings of the 6th International Conference on Automotive User Interfaces and Interactive Vehicular Applications*, Seattle, WA. ACM, 1–3. DOI: https://doi.org/10.1145/2667239.2667311.

M. Ringel, E. Cutrell, S. Dumais, and E. Horvitz. September. 2003. Milestones in time: The value of landmarks in retrieving information from personal stores. In *Proceedings of IFIP INTERACT03: Human–Computer Interaction*, Zürich, Switzerland. IOS Press, 184–191.

G. Robertson, A. Newell, and K. Ramakrishna. August. 1977. *ZOG: A Man–Machine Communication Philosophy*. Carnegie Mellon University Technical Report. https://doi.org/10.1184/R1/6612989.v1.

G. G. Robertson, S. K. Card, and J. D. Mackinlay. 1993. Information visualization using 3D interactive animation. *Commun. ACM* 36, 4, 57–71. DOI: https://doi.org/10.1145/255950.153577.

P. Romich. 2021. Our History. PRC-Saltillo, Inc. Retrieved February 8, 2021 from https://www.prc-saltillo.com/our-history.

P. A. Romitti, Y. Zhu, S. Puzhankara, K. A. James, S. K. Nabukera, G. K. Zamba, E. Ciafaloni, C. Cunniff, C. M. Druschel, K. D. Mathews, D. J. Matthews, F. J. Meaney, J. G. Andrews, K. M. Conway, D. J. Fox, N. Street, M. M. Adams, and J. Bolen. March. 2015. Prevalence of Duchenne and Becker muscular dystrophies in the United States. *Pediatrics* 135, 3, 513–521. DOI: https://doi.org/10.1542/peds.2014-2044, https://pubmed.ncbi.nlm.nih.gov/25687144/.

A. Rose. April. 1946. Lightning strikes mathematics. *Popular Science* 148, 4, 83–86. Image from https://en.wikipedia.org/wiki/ENIAC#/media/File:Classic_shot_of_the_ENIAC.jpg.

M. Roseman and S. Greenberg. 1996. Building real time groupware with GroupKit, a groupware toolkit. *ACM Trans. Comput. Hum. Interact.* 3, 1, 66–106. DOI: https://doi.org/10.1145/226159.226162.

B. M. Rosen. July 11. 1979. VisiCalc: Breaking the Personal Computer Software Bottleneck. *Morgan Stanley Electronics Letter.* http://www.bricklin.com/history/rosenletter.htm.

D. Ross and J. Rodriguez. 1963. Theoretical foundations for the computer-aided design system. In *Proceedings of the May 21–23, 1963, Spring Joint Computer Conference*. ACM, 305–322. DOI: https://doi.org/10.1145/1461551.1461589.

S. Ruan, J. O. Wobbrock, K. Liou, A. Ng, and J. A. Landay. 2018. Comparing speech and keyboard text entry for short messages in two languages on touchscreen phones. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 1, 4, 1–23. DOI: https://doi.org/10.1145/3161187.

D. Rubine. July. 1991a. Specifying gestures by example. *ACM SIGGRAPH Comput. Graph*. 25, 4, 329–337. DOI: https://doi.org/10.1145/127719.122753.

D. Rubine. 1991b. Integrating gesture recognition and direct manipulation. *Summer 91 USENIX Technical Proceedings*. Also CMU ITC Technical Report, CMU-ITC-91-100. http://ra.adm.cs.cmu.edu/anon/usr0/ftp/usr/anon/itc/CMU-ITC-100.pdf.

D. Rubine. 1991c. *The Automatic Recognition of Gestures*. School of Computer Science Technical Report, CMU-CS-91-202. Carnegie Mellon University. http://reports-archive.adm.cs.cmu.edu/anon/1991/CMU-CS-91-202.ps.

M. Ruoff, B. A. Myers, and A. Maedche. 2023. ONYX: Assisting users in teaching natural language interfaces through multi-modal interactive task learning. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, Hamburg, Germany. ACM, 1–16. DOI: https://doi.org/10.1145/3544548.3580964.

J. Rutledge and T. Selker. 1990. In-keyboard analog pointing device: A case for the pointing stick. In *Technical Video Program of the CHI'90 Conference. SIGGRAPH Video Review* 1, 55. https://www.youtube.com/watch?v=TKGMIKGnm88.

D. Saffer. 2013. *Microinteractions: Designing with Details*. O'Reilly Media, Inc.

D. Salber, A. K. Dey, and G. D. Abowd. 1999. The context toolkit: Aiding the development of context-enabled applications. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '99)*, Pittsburgh, PA. ACM, 434–441. DOI: https://doi.org/10.1145/302979.303126.

P. H. Salus. 2005. Chapter 4. A tale of two editors. *The Daemon, the Gnu and the Penguin*. Groklaw. http://www.groklaw.net/article.php?story=20050422235450910.

M. Sannella. November. 1994. SkyBlue: A multi-way local propagation constraint solver for user interface construction. In *Proceedings of the 7th Annual ACM Symposium on User Interface Software and Technology (UIST '94)*, Marina del Rey, CA. ACM, 137–146. DOI: https://doi.org/10.1145/192426.192485.

A. L. Santos and B. A. Myers. April. 2017. Design annotations to improve API discoverability. *J. Syst. Softw.* 126, 17–33. DOI: https://doi.org/10.1016/j.jss.2016.12.036.

C. Scaffidi, B. Myers, and M. Shaw. May 10–18. 2008. Topes: Reusable abstractions for validating data. In *Proceedings of the 30th International Conference on Software Engineering (ICSE '08)*, Leipzig, Germany. ACM, 1–10. DOI: https://doi.org/10.1145/1368088.1368090.

R. W. Scheifler and J. Gettys. April. 1986. The X window system. *ACM Trans. Graph.* 5, 2, 79–109. DOI: https://doi.org/10.1145/22949.24053.

K. J. Schmucker. August. 1986. MacApp: An application framework. *Byte* 11, 8, 189–193.

R. Scott. 1984. 1984—The original Apple Macintosh commercial. Video. https://www.youtube.com/watch?v=VtvjbmoDx-I.

J. Sculley. 1989. The relationship between business and higher education: A perspective on the 21st century. *Commun. ACM* 32, 9, 1056–1061. https://doi.org/10.1145/66451.66452.

A. Sears and B. Shneiderman. 1994. Split menus: Effectively using selection frequency to organize menus. *ACM Trans. Comput. Hum. Interact.* 1, 1, 27–51. DOI: https://doi.org/10.1145/174630.174632.

J. Seymour. March 15. 1994. Everything old is new again. *PC Magazine*, 99–100.

S. Sharmin, O. Spakov, and K.-J. Raiha. 2013. Reading on-screen text with gaze-based auto-scrolling. In *Proceedings of the 2013 Conference on Eye Tracking South Africa (ETSA '13)*, Cape Town, South Africa. ACM, 24–31. DOI: https://doi.org/10.1145/2509315.2509319.

B. Sheil. February. 1983. Environments for exploratory programming. In B. A. Sheil and L. M. Masinter (Eds.), *Datamation.* Reprinted in *Papers on Interlisp-D*, Xerox PARC Tech Report CIS-5.

K. Sherwin. October 26. 2014. Progress indicators make a slow system less insufferable. *Nielsen Norman Group*. https://www.nngroup.com/articles/progress-indicators/.

M. Shilman, D. S. Tan, and P. Simard. 2006. CueTIP: A mixed-initiative interface for correcting handwriting errors. In *Proceedings of the 19th Annual ACM Symposium on User Interface Software and Technology (UIST '06)*, Montreux, Switzerland. ACM, 323–332. DOI: https://doi.org/10.1145/1166253.1166304.

K. Shima, K. Onishi, R. Takada, T. Adachi, B. Shizuki, and J. Tanaka. 2016. Investigating accuracy of tilting operation on wrist-worn devices with touchscreens. In *Proceedings of the 2016 CHI Conference Extended Abstracts on Human Factors in Computing Systems*, San Jose, California. ACM, 2705–2711. DOI: https://doi.org/10.1145/2851581.2892377.

B. Shneiderman. August. 1983. Direct manipulation: A step beyond programming languages. *IEEE Comput.* 16, 8, 57–69. DOI: https://doi.org/10.1109/MC.1983.1654471.

B. Shneiderman. 1987. *Designing the User Interface: Strategies for Effective Human–Computer Interaction* (1st. ed.). Addison-Wesley, Reading, MA.

B. Shneiderman. 2022. *Human-Centered AI*. Oxford University Press, Oxford, UK.

B. Shneiderman and P. Maes. 1997. Direct manipulation vs. interface agents. *Interactions* 4, 6, 42–61. DOI: https://doi.org/10.1145/267505.267514.

B. Shneiderman, C. Plaisant, M. Cohen, S. Jacobs, N. Elmqvist, and N. Diakopoulos. 2017. *Designing the User Interface: Strategies for Effective Human–Computer Interaction* (6th. ed.). Pearson, Boston.

M. Silfverberg. 2003. Using mobile keypads with limited visual feedback: Implications to handheld and wearable devices. In *Proceedings of the Mobile HCI 2003*, Vol. 2795: Lecture Notes in Computer Science. Springer, Berlin, 76–90. DOI: https://doi.org/10.1007/978-3-540-45233-1_7.

D. C. Smith. 1977. *Pygmalion: A Computer Program to Model and Stimulate Creative Thought*. Birkhauser Verlag, Basel. Ph.D. thesis. Stanford University Computer Science Department, 1975.

R. B. Smith. 1986. The Alternate Reality Kit: An animated environment for creating interactive simulations. In *Proceedings of the 1986 IEEE Computer Society Workshop on Visual Languages (VL '86)*. IEEE, 99–106.

R. B. Smith. 1987. Experiences with the Alternate Reality Kit: An example of the tension between literalism and magic. *IEEE Comput. Graph. Appl.* 7, 9, 42–50. DOI: https://doi.org/10.1109/MCG.1987.277078.

T. Smith. June 28. 2006. Palm pays $22.5m to end Xerox patent fight. *The Register*. https://www.theregister.co.uk/2006/06/28/palm_xerox_settlement/.

S. L. Smith and J. N. Mosier. August. 1986. *Guidelines for Designing User Interface Software*. MITRE, Bedford, MA. http://hcibib.org/sam/.

D. C. Smith, C. Irby, R. Kimball, and E. Harslem. June 7–10. 1982a. The Star user interface: An overview. In *Proceedings of the 1982 National Computer Conference (AFIPS '82)*, Houston, TX. ACM, 515–528. DOI: https://doi.org/10.1145/1500774.1500840.

D. C. Smith, C. Irby, R. Kimball, B. Verplank, and E. Harslem. April. 1982b. Designing the Star user interface. *Byte* 7, 4, 242–282. DOI: https://doi.org/10.1007/978-1-4757-3510-9_21.

D. C. Smith, A. Cypher, and J. Spohrer. July. 1994. KidSim: Programming agents without a programming language. *Commun. ACM* 37, 7, 54–67. DOI: https://doi.org/10.1145/176789.176795.

S. S. Snibbe, K. P. Herndon, D. C. Robbins, D. B. Conner, and A. van Dam. 1992. Using deformations to explore 3D widget design. *ACM SIGGRAPH Comput. Graph*. 26, 2, 351–352. DOI: https://doi.org/10.1145/133994.134091.

R. Soukoreff and I. MacKenzie. March 31. 2001. Measuring errors in text entry tasks: An application of the Levenshtein string distance statistic. In *Proceedings of the CHI '01 Extended Abstracts on Human Factors in Computing Systems (CHI EA '01)*, Seattle, WA. ACM, 319–320. DOI: https://doi.org/10.1145/634067.634256.

R. W. Soukoreff and I. S. MacKenzie. December 1. 2004. Towards a standard for pointing device evaluation, perspectives on 27 years of Fitts' law research in HCI. *Int. J. Hum. Comput. Stud.* 61, 6, 751–789. DOI: https://doi.org/10.1016/j.ijhcs.2004.09.001, http://www.sciencedirect.com/science/article/pii/S1071581904001016.

R. Spence. January. 1976. Human factors in interactive graphics. *Comput. Aided Des.* 8, 1, 49–53. DOI: https://doi.org/10.1016/0010-4485(76)90009-9.

Sprint Nextel. 2006. *Getting Started With Your BlackBerry® 7250™, Powered by Sprint PCS*. https://www.blackberryservice.es/manuales/manual-blackberry-7250-ingles.pdf.

R. M. Stallman. August. 1979. *EMACS: The Extensible, Customizable, Self-Documenting Display Editor*. MIT Artificial Intelligence Lab, AI Memo 519a. http://hdl.handle.net/1721.1/5736.

T. Starner, J. Weaver, and A. Pentland. 1998. Real-time American sign language recognition using desk and wearable computer based video. *IEEE Trans. Pattern Anal. Mach. Intell.*, 20, 12, 1371–1375. DOI: https://doi.org/10.1109/34.735811.

G. L. Steele Jr. (Ed.). 1984. *Common Lisp: The Language*. Digital Press.

Stenograph L.L.C. 2022. The history of writing machines. *Stenograph*. https://www.stenograph.com/history-writers.

A. Stevens, B. Roberts, and L. Stead. March/April. 1983. The use of a sophisticated graphics interface in computer-assisted instruction. *IEEE Comput. Graph. Appl.* 3, 2, 25–31. DOI: https://doi.org/10.1109/MCG.1983.262988.

D. M. Stoffel. September. 1982. Talking terminals. *Byte* 7, 9, 218–227. https://web.archive.org/web/20060625225004/, http://www.edstoffel.com/david/talkingterminals.html.

J. Stylos, B. A. Myers, and A. Faulring. October 24–27. 2004. Citrine: Providing intelligent copy-and-paste. In *Proceedings of the 17th Annual ACM Symposium on User Interface Software and Technology (UIST '04)*, Santa Fe, NM. ACM, 185–188. DOI: https://doi.org/10.1145/1029632.1029665.

B. Suhm, B. Myers, and A. Waibel. October. 1996. Interactive recovery from speech recognition errors in speech user interfaces. In *Proceedings of the 4th International Conference on Spoken Language Processing (ICSLP '96)*, Vol. 2. IEEE, 865–868. DOI: https://doi.org/10.1109/ICSLP.1996.607738.

N. Sukaviriya, S. Kovacevic, J. Foley, B. Myers, D. Olsen, and M. Schneider-Hufschmidt. November. 1994. Model-based user interfaces: What is it and why should we care? In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology (UIST '94)*, Marina del Rey, CA. ACM, 133–135. DOI: https://doi.org/10.1145/192426.192479.

D. Sullivan. August 25. 2008. Google.com finally gets Google Suggest feature. *Search Engine Land*. https://searchengineland.com/googlecom-finally-gets-google-suggest-feature-14626.

Sun. October. 1986. *NeWS Preliminary Technical Overview*. Sun Microsystems, Inc.

I. E. Sutherland. 1963. Sketchpad: A man–machine graphical communication system. In *Proceedings of the May 21–23 1963, Spring Joint Computer Conference (AFIPS '63)*. ACM, 329–346. DOI: https://doi.org/10.1145/1461551.1461591.

I. Sutherland. 1964. Computer Sketchpad; 20:42 min Video. mpg: http://www.ibiblio.org/openvideo/video/chi/chi84_15_m2.mpg, or in two parts: http://www.youtube.com/watch?v=USyoT_Ha_bA and http://www.youtube.com/watch?v=BKM3CmRqK2o.

W. R. Sutherland. 1966. *On-line Graphical Specification of Computer Procedures*. Ph.D. thesis. Massachusetts Institute of Technology. Lincoln Labs Report TR-405.

I. E. Sutherland. 1968. A head-mounted three dimensional display. In *Proceedings of the December 9–11, 1968, Fall Joint Computer Conference, Part I (AFIPS '68)*, San Francisco, CA. ACM, 757–764. DOI: https://doi.org/10.1145/1476589.1476686.

D. C. Swinehart. 1974. *Copilot: A Multiple Process Approach to Interactive Programming Systems*. Ph.D. thesis. Computer Science Department, Stanford University. SAIL Memo AIM-230 and CSD Report STAN-CS-74-412.

D. Swinehart, P. Zellweger, R. Beach, and R. Hagmann. October. 1986. A structural view of the cedar programming environment. *ACM Trans. Program. Lang. Syst*. 8, 4, 419–490. DOI: https://doi.org/10.1145/6465.6466.

P. A. Szekely and B. A. Myers. November. 1988. A user interface toolkit based on graphical objects and constraints. *ACM SIGPLAN Not.* 23, 11, 36–45. DOI: https://doi.org/10.1145/62084.62088.

P. Szekely, P. Luo, and R. Neches. April. 1993. Beyond interface builders: Model-based interface tools. In *Proceedings of the INTERCHI '93 Conference on Human Factors in*

Computing Systems (INTERCHI '93), Amsterdam, Netherlands. ACM, 383–390. DOI: https://doi.org/10.5555/164632.164954.

Tap Systems Inc. The Tap Strap. 2019. See also archive at: https://web.archive.org/web/20230606165812/https://www.tapwithus.com/shop/.

E. A. Taub. September 27. 2018. 2-Second rule for distracted driving can mean life or death. *New York Times*. Section B, 5. https://www.nytimes.com/2018/09/27/business/distracted-driving-auto-industry.html.

T. Teitelbaum and T. Reps. 1981. The Cornell program synthesizer: A syntax-directed programming environment. *Commun. ACM* 24, 9, 563–573. DOI: https://doi.org/10.1145/358746.358755.

W. Teitelman. 1966. *Pilot: A Step Toward Man–Computer Symbiosis*. Ph.D. Dissertation, Massachusetts Institute of Technology. Number MAC-TR-32.

W. Teitelman. 1979. March 8. 1977. A display oriented programmer's assistant. *Int. J. Man Mach. Stud*. 11, 2, 157–187. DOI: https://doi.org/10.1016/S0020-7373(79)80015-2. Also Xerox PARC Technical Report CSL-77-3, Palo Alto, CA.

W. Teitelman. 1985. A tour through cedar. *IEEE Trans. Softw Eng.* 11, 3, 285–302. DOI: https://doi.org/10.1109/TSE.1985.232212.

W. Teitelman. 2008. History of Interlisp. In *Celebrating the 50th Anniversary of Lisp (LISP50)*, Nashville, TN. ACM, 1–5. DOI: https://doi.org/10.1145/1529966.1529971.

L. Terveen, W. Hill, and B. Amento. 1998. Collaborative filtering to locate, comprehend, and organize collections of Web sites. *ACM SIGART Bull*. 9, 3–4, 10–17. DOI: https://doi.org/10.1145/302342.302344.

L. Tesler. August. 1981. The Smalltalk environment. *Byte* 6, 8, 90–147.

L. Tesler. 2011. What's it like to design a user interface for six billion people? (video of Lifetime Achievement in Practice Award talk). In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '11)*, Vancouver, BC. ACM (with an introduction by Brad Myers). DOI: https://doi.org/10.1145/1978942.2167167.

L. Tesler. 2012. A personal history of modeless text editing and cut/copy-paste. *Interactions* 19, 4, 70–75. DOI: https://doi.org/10.1145/2212877.2212896.

P. Thurrott and R. Rivera. 2009. *Windows 7 Secrets*. Wiley.

D. Tilbrook. 1976. *A Newspaper Pagination System*. Masters's thesis. Department of Computer Science, University of Toronto.

B. Tolliver. March. 1965. TVEdit. Stanford Time Sharing Memo 32.

Trace. 2021. *History*. Trace Research & Development Center, University of Maryland. https://trace.umd.edu/history/.

I. Tucker. July 26. 2020. Roger McNamee: "Facebook is a threat to whatever remains of democracy in the US." *The Guardian*. https://www.theguardian.com/technology/2020/jul/26/roger-mcnamee-facebook-is-a-threat-to-whatever-remains-of-democracy-in-the-us.

J. Underkoffler and H. Ishii. 1999. Urp: A luminous-tangible workbench for urban planning and design. In *Proceedings of the SIGCHI Conference on Human Factors in Computing*

Systems (CHI '99), Pittsburgh, PA. ACM, 386–393. DOI: https://doi.org/10.1145/302979.303114.

A. van Dam. 1988. Hypertext '87: Keynote address. *Commun. ACM* 31, 7, 887–895. DOI: https://doi.org/10.1145/48511.48519.

A. van Dam, S. Carmody, T. Gross, T. Nelson, and D. Rice. 1969. A hypertext editing system for the 360. In *Proceedings of the Conference in Computer Graphics*. University of Illinois.

D. K. van Duyne, J. A. Landay, and J. I. Hong. 2002. *The Design of Sites: Patterns, Principles, and Processes for Crafting a Customer-Centered Web Experience*. Addison-Wesley.

G. C. Vanderheiden. October. 2008. Ubiquitous accessibility, common technology core, and micro assistive technology: Commentary on "Computers and people with disabilities". *ACM Trans. Access. Comput.* 1, 2, 1–7. DOI: https://doi.org/10.1145/1408760.1408764.

R.-D. Vatavu, L. Anthony, and J. O. Wobbrock. 2012. Gestures as point clouds: A $P recognizer for user interface prototypes. In *Proceedings of the 14th ACM International Conference on Multimodal Interaction (ICMI '12)*, Santa Monica, CA. ACM, 273–280. DOI: https://doi.org/10.1145/2388676.2388732.

M. Vázquez, E. Brockmeyer, R. Desai, C. Harrison, and S. E. Hudson. 2015. 3D printing pneumatic device controls with variable activation force capabilities. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems (CHI '15)*, Seoul, Republic of Korea. ACM, 1295–1304. DOI: https://doi.org/10.1145/2702123.2702569.

M. M. Veloso. March. 2002. Entertainment robotics. *Commun. ACM* 45, 3, 59–63. DOI: https://doi.org/10.1145/504729.504755.

V. Vinge. December 1. 1993. The coming technological singularity: How to survive in the post-human era. Presented at the *VISION-21 Symposium* sponsored by NASA Lewis Research Center, March 30–31, 1993. NTRS—NASA Technical Reports Server. https://ntrs.nasa.gov/citations/19940022856.

Visix Software Inc. 1997. *Galaxy Application Environment.* (Company dissolved in 1998. Galaxy was bought by Ambiencia Information Systems, Inc., Campinas, Brazil, support@ambiencia.com. http://www.ambiencia.com).

J. S. Vitter. 1984. US&R: A New Framework for redoing (Extended Abstract). ACM *SIGSOFT Softw. Eng. Notes* 9, 3, 168–176. DOI: https://doi.org/10.1145/390010.808262.

J. M. Vlissides and M. A. Linton. November. 1989. Unidraw: A framework for building domain-specific editors. In *Proceedings of the 2nd Annual ACM SIGGRAPH Symposium on User Interface Software and Technology (UIST '89)*, Williamsburg, VA. ACM, 158–167. DOI: https://doi.org/10.1145/73660.73680.

L. von Ahn, M. Blum, and J. Langford. 2004. Telling humans and computers apart automatically. *Commun. ACM* 47, 2, 56–60. DOI: https://doi.org/10.1145/966389.966390.

A. R. Wagner, J. Borenstein, and A. Howard. 2018. Overtrust in the robotic age. *Commun. ACM* 61, 9, 22–24. DOI: https://doi.org/10.1145/3241365.

J. Wang, S. Zhai, and J. Canny. 2006. Camera phone based motion sensing: Interaction techniques, applications and performance study. In *Proceedings of the 19th Annual ACM*

Symposium on User Interface Software and Technology (UIST '06), Montreux, Switzerland. ACM, 101–110. DOI: https://doi.org/10.1145/1166253.1166270.

R. Want, B. N. Schilit, N. Adams, R. Gold, K. Petersen, D. Goldberg, J. R. Ellis, and M. Weiser. December. 1995. An overview of the PARCTAB ubiquitous computing experiment. *IEEE Pers. Commun.* 2, 6, 28–43. Also appears as Xerox PARC Technical Report CSL-95-1, March. 1995.

J. R. Ward and B. Blesser. 1985. Interactive recognition of handprinted characters for computer input. *IEEE Comput. Graph. Appl.* 5, 9, 24–37. DOI: https://doi.org/10.1109/98.475986.

D. J. Ward, A. F. Blackwell, and D. J. C. MacKay. 2000. Dasher—A data entry interface using continuous gestures and language models. In *Proceedings of the 13th Annual ACM Symposium on User Interface Software and Technology (UIST '00)*, San Diego, CA. ACM, 129–137. DOI: https://doi.org/10.1145/354401.354427.

C. Ware and D. R. Jessome. 1988. Using the bat: A six-dimensional mouse for object placement. *IEEE Comput. Graph. Appl.* 8, 6, 65–70. DOI: https://doi.org/10.1109/38.20319.

P. Warman. November 2. 2018. Newzoo cuts global games forecast for 2018 to $134.9 billion; Lower mobile growth partially offset by very strong growth in console segment. *Newzoo*. https://newzoo.com/insights/articles/newzoo-cuts-global-games-forecast-for-2018-to-134-9-billion/.

J. E. Warnock. July–September. 2018. The origins of PostScript. *IEEE Ann. Hist. Comput.* 40, 3, 68–76. DOI: https://doi.org/10.1109/MAHC.2018.033841112.

M. Weiser. September. 1991. The computer for the 21st century. *Sci. Am.* 265, 3, 94–104. Reprinted in *SIGMOBILE Mob. Comput. Commun. Rev.* 3, 3 (July 1999), 3–11. http://doi.org/10.1145/329124.329126.

M. Weiser. July. 1993. Some computer science issues in ubiquitous computing. *Commun. ACM* 36, 7, 74–83. DOI: http://doi.org/10.1145/159544.159617.

J. Weizenbaum. 1976. *Computer Power and Human Reason: From Judgment to Calculation*. W. H. Freeman, New York. https://dl.acm.org/doi/10.5555/540249.

P. Wellner. 1991. The DigitalDesk calculator: Tangible manipulation on a desk top display. In *Proceedings of the 4th Annual ACM Symposium on User Interface Software and Technology (UIST '91)*, Hilton Head, SC. ACM, 27–33. DOI: https://doi.org/10.1145/120782.120785.

K. Whitenton. March 11. 2018. The two UX gulfs: Evaluation and execution. *Nielsen Norman Group*. https://www.nngroup.com/articles/two-ux-gulfs-evaluation-execution/.

WHO. October 8. 2019. *WHO Launches First World Report on Vision*. World Health Organization. https://www.who.int/news/item/08-10-2019-who-launches-first-world-report-on-vision.

C. Wiecha, W. Bennet, S. Boies, and J. Gould. March. 1989. Generating highly interactive user interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (SIGCHI '89)*, Austin, TX. ACM, 277–282. DOI: https://doi.org/10.1145/67449.67502.

C. Wiecha, W. Bennett, S. Boies, J. Gould, and S. Greene. July. 1990. ITS: A tool for rapidly developing interactive applications. *ACM Trans. Inf. Syst.* 8, 3, 204–236. DOI: https://doi.org/10.1145/98188.98194.

G. Williams. February. 1984. The Apple Macintosh computer. *Byte* 9, 2, 30–54.

M. Williams. 2013. Jury orders Samsung to pay $290M to Apple in patent case. *ComputerWorld*. IDG. https://www.computerworld.com/article/2486197/technology-law-regulation/jury-orders-samsung-to-pay–290m-to-apple-in-patent-case.html.

C. Williamson and B. Shneiderman. June 21–24. 1992. The dynamic HomeFinder: Evaluating dynamic queries in a real-estate information exploration system. In *Proceedings of the 15th Annual International ACM SIGIR Conference on Research and Development in Information Retrieva (SIGIR '92)*, Copenhagen. ACM, 338–346. Also tech report HCIL-92-01. DOI: https://doi.org/10.1145/133160.133216. https://hcil.umd.edu/pub-perm-link/?id=92-01.

A. Wilson and S. Shafer. 2003. XWand: UI for intelligent spaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '03)*, Ft. Lauderdale, FL. ACM, 545–552. DOI: https://doi.org/10.1145/642611.642706.

A. D. Wilson and R. Sarin. 2007. BlueTable: Connecting wireless mobile devices on interactive surfaces using vision-based handshaking. In *Proceedings of Graphics Interface 2007 (GI '07)*, Montreal, QC. ACM, 119–125. DOI: https://doi.org/10.1145/1268517.1268539.

T. Winograd. January. 1972. Understanding natural language. *Cogn. Psychol.* 3, 1, 1–191. DOI: https://doi.org/10.1016/0010-0285(72)90002-3, https://www.sciencedirect.com/science/article/pii/0010028572900023.

Wired News. October 5. 1998. Computer Intelligence with Fur. *WIRED*. https://www.wired.com/1998/10/computer-intelligence-with-fur/.

N. E. Wiseman, H. U. Lemke, and J. O. Hiles. 1969. PIXIE: A new approach to graphical man–machine communication. In *Proceedings of the 1969 IEEE CAD Conference*, Southampton. IEEE, 463–471.

J. O. Wobbrock and B. A. Myers. 2006. Analyzing the input stream for character-level errors in unconstrained text entry evaluations. *ACM Trans. Comput. Hum. Interact.* 13, 4, 458–489. DOI: https://doi.org/10.1145/1188816.1188819.

J. O. Wobbrock, J. Forlizzi, S. E. Hudson, and B. A. Myers. October 27–30. 2002. WebThumb: Interaction techniques for small-screen browsers. In *Proceedings of the CHI Letters: ACM Symposium on User Interface Software and Technology (UIST '02)*, Paris, France. ACM, 205–208. DOI: https://doi.org/10.1145/571985.572014.

J. O. Wobbrock, B. A. Myers, and J. Kembel. November 2–5. 2003. EdgeWrite: A stylus-based text entry method designed for high accuracy and stability of motion. In *Proceedings of the 16th Annual ACM Symposium on User Interface Software and Technology (UIST '03)*, Vancouver, BC. ACM, 61–70. DOI: https://doi.org/10.1145/964696.964703.

J. O. Wobbrock, B. A. Myers, and H. H. Aung. May 17–19. 2004. Writing with a joystick: A comparison of date stamp, selection keyboard, and EdgeWrite. In *Proceedings of the Graphics Interface 2004 (GI '04)*, London, ON. Canadian Human–Computer

Communications Society, 1–8. https://kilthub.cmu.edu/articles/journal_contribution/Writing_With_A_Joystick_A_Comparison_of_Date_Stamp_Selection_Keyboard_and_EdgeWrite/6470696/1. DOI: https://doi.org/10.1184/R1/6626567.v1.

J. O. Wobbrock, H. H. Aung, B. Rothrock, and B. A. Myers. April 2–7. 2005. Maximizing the guessability of symbolic input. In *Proceedings of the CHI '05 Extended Abstracts on Human Factors in Computing Systems (CHI EA '05)*, Portland, OR. ACM, 1869–1872. DOI: https://doi.org/10.1145/1056808.1057043.

J. O. Wobbrock, B. A. Myers, and D. H. Chau. 2006. In-stroke word completion. In *Proceedings of the 19th Annual ACM Symposium on User Interface Software and Technology (UIST '06)*, Montreux, Switzerland. ACM, 333–336. DOI: https://doi.org/10.1145/1166253.1166305.

J. O. Wobbrock, A. D. Wilson, and Y. Li. 2007. Gestures without libraries, toolkits or training: A $1 recognizer for user interface prototypes. In *Proceedings of the 20th Annual ACM Symposium on User Interface Software and Technology (UIST '07)*, Newport, RI. ACM, 159–168. DOI: https://doi.org/10.1145/1294211.1294238.

J. O. Wobbrock, M. R. Morris, and A. D. Wilson. 2009. User-defined gestures for surface computing. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, Boston, MA. ACM, 1083–1092. DOI: https://doi.org/10.1145/1518701.1518866.

D. Wolber. December. 1997. An interface builder for designing animated interfaces. *ACM Trans. Comput. Hum. Interact.* 4, 4, 347–386. DOI: https://doi.org/10.1145/267135.267142.

WordCounter. June 2. 2016. How fast does the average person speak? *WordCounter*.net. https://wordcounter.net/blog/2016/06/02/101702_how-fast-average-person-speaks.html.

Xerox (Ed.). 1978. *Alto User's Handbook.* Xerox Palo Alto Research Center, Palo Alto, CA. http://history-computer.com/Library/AltoUsersHandbook.pdf.

Xerox. October. 1983. *Interlisp Reference Manual*. Xerox Corporation, Pasadena, CA.

H. Xia, B. Araujo, T. Grossman, and D. Wigdor. 2016. Object-oriented drawing. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems (CHI '16)*, San Jose, CA. ACM, 4610–4621. DOI: https://doi.org/10.1145/2858036.2858075.

R. Xiao, G. Laput, and C. Harrison. 2014. Expanding the input expressivity of smartwatches with mechanical pan, twist, tilt and click. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '14)*, Toronto, ON. ACM, 193–196. DOI: https://doi.org/10.1145/2556288.2557017.

R. Xiao, J. Schwarz, and C. Harrison. 2015. Estimating 3D finger angle on commodity touchscreens. In *Proceedings of the 2015 International Conference on Interactive Tabletops & Surfaces (ITS '15)*, Madeira, Portugal. ACM, 47–50. DOI: https://doi.org/10.1145/2817721.2817737.

Y. Yang. 1988. Undo support models. *Int. J. Man Mach. Stud.* 28, 5, 457–481. DOI: https://doi.org/10.1016/S0020-7373(88)80056-7.

H.-D. Yang. 2015. Sign language recognition with the Kinect sensor based on conditional random fields. *Sensors* 15, 1, 135–147. DOI: https://doi.org/10.3390/s150100135.

J. (Junrui) Yang, M. S. Lam, and J. A. Landay. October. 2020. DoThisHere: Multimodal interaction to improve cross-application tasks on mobile devices. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology (UIST '20)*. ACM, 35–44. DOI: https://doi.org/10.1145/3379337.3415841.

H. Yang, T. Johnson, K. Zhong, D. Patel, G. Olson, C. Majidi, M. Islam, and L. Yao. 2022. ReCompFig: Designing dynamically reconfigurable kinematic devices using compliant mechanisms and tensioning cables. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems (CHI '22)*, New Orleans, LA. ACM, 170. DOI: https://doi.org/10.1145/3491102.3502065.

Y. S. Yoon and B. A. Myers. May 16–24. 2015. Supporting selective undo in a code editor. In *Proceedings of the 37th International Conference on Software Engineering (ICSE '15)*, Florence, Italy, Vol. 1. IEEE, 223–233. DOI: https://doi.org/10.1109/ICSE.2015.43.

W. Yoshizaki, Y. Sugiura, A. C. Chiou, S. Hashimoto, M. Inami, T. Igarashi, Y. Akazawa, K. Kawachi, S. Kagami, and M. Mochimaru. 2011. An actuated physical puppet as an input device for controlling a digital manikin. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '11)*, Vancouver, BC. ACM, 637–646. DOI: https://doi.org/10.1145/1978942.1979034.

B. V. Zanden and B. A. Myers. April. 1990. Automatic, look-and-feel independent dialog creation for graphical user interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '90)*, Seattle, WA. ACM, 27–34. DOI: https://doi.org/10.1145/97243.97248.

B. V. Zanden, B. A. Myers, D. Giuse, and P. Szekely. November. 1991. The importance of pointer variables in constraint models. In *Proceedings of the 4th Annual ACM Symposium on User Interface Software and Technology (UIST '91)*, Hilton Head, SC. ACM, 155–164. DOI: https://doi.org/10.1145/120782.120799.

J. Zawinski. 1999. Emacs Timeline. https://www.jwz.org/doc/emacs-timeline.html.

S. Zhai and P.-O. Kristensson. 2003. Shorthand writing on stylus keyboard. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, Ft. Lauderdale, FL. ACM, 97–104. DOI: https://doi.org/10.1145/642611.642630.

S. Zhai, B. A. Smith, and T. Selker. July 14–18. 1997. Improving browsing performance: A study of four input devices for scrolling and pointing. In *Proceedings of the IFIP TC13 International Conference on Human–Computer Interaction (INTERACT '97)*, Sydney, Australia. Springer, 286–292. DOI: https://doi.org/10.1007/978-0-387-35175-9_48.

S. Zhai, M. Hunter, and B. A Smith. November 6–8. 2000. The Metropolis keyboard—An exploration of quantitative techniques for virtual keyboard design. In *Proceedings of the 13th Annual ACM Symposium on User Interface Software and Technology (UIST '00)*. San Diego, CA. ACM, 119–128. DOI: https://doi.org/10.1145/354401.354424.

S. Zhai, P. O. Kristensson, P. Gong, M. Greiner, S. A. Peng, L. M. Liu, and A. Dunnigan. 2009. ShapeWriter on the iPhone: From the laboratory to the real world. In *Proceedings of the CHI '09 Extended Abstracts on Human Factors in Computing Systems (CHI EA '09)*, Boston, MA. ACM, 2667–2670. DOI: https://doi.org/10.1145/1520340.1520380.

Z. Zhang. 2012. Microsoft Kinect sensor and its effect. *IEEE MultiMed.* 19, 2, 4–10. DOI: https://doi.org/10.1109/MMUL.2012.24.

Q. Zhou, G. Fitzmaurice, and F. Anderson. 2022. In-depth mouse: Integrating desktop mouse into virtual reality. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*, New Orleans, LA. ACM, 354. DOI: https://doi.org/10.1145/3491102.3501884.

B. Zimmer. September 15. 2009. The age of undoing. *New York Times Magazine*. https://www.nytimes.com/2009/09/20/magazine/20FOB-onlanguage-t.html?_r=0.

T. G. Zimmerman, J. Lanier, C. Blanchard, S. Bryson, and Y. Harvill. 1986. A hand gesture interface device. In *Proceedings of the SIGCHI/GI Conference on Human Factors in Computing Systems and Graphics Interface*, Toronto, ON. ACM, 189–192. DOI: https://doi.org/10.1145/29933.275628.

S. Zuberec. 2000. Chapter 5: The interaction design of Microsoft Windows CE. In E. Bergman (Ed.), *Information Appliances and Beyond.* Morgan Kaufmann, San Francisco, 103–129.

S. Zuboff. April 4. 2015. Big other: Surveillance capitalism and the prospects of an information civilization. *J. Inf. Technol.* 30, 1, 75–89. DOI: https://doi.org/10.1057/jit.2015.5, https://ssrn.com/abstract=2594754.

Index

2½ dimensions, 263
3D. *See* Three dimensions
3D touch, 93, 109
6DOF, 329
802.11, 73

A/B testing. *See* Controlled experiment
Abort. *See* Cancel
Abort(ing), 16, 121
Absolute coordinates, 106
Absolute layout, 492–493
Accelerator keys. *See* Keyboard shortcuts
Accelerometer, 639
Accessibility, 368, 379, 424, 528, 632–633, 635–636
Accessible, 424–425
Accordion Menu, 167
Accuracy, 229–230, 379
  of gestures, 378–379
  of pointing, 110
  of popular methods, 232
  required level of, 459
  testing, 110–115
  of touch, 102
ACM Transactions on Computer–Human Interaction, 627
ACT-R, 479
ActionScript, 47
Active window. *See* Input focus
Adaptive input devices, 429–430
Additional parameters, 367
Adobe, 261
  Acrobat Pro DC, 142, 297
  Adobe Illustrator, 5
  Illustrator, 261
  Photoshop, 506
  XD, 506
Advanced Research Project Agency (ARPA), 51
Adventure game, 54
Aero Shake, 312
Aero Snap (Windows 7), 312
Affordance, 6, 17, 133, 367, 442, 444, 461, 526
  lack of, 367
AI models, 467
Airtap, 642
Alexa (Amazon), 441, 444
Algorithmic bias, 459
Alias and Alias|Wavefront, 289, 510, 560
Alice Corp. v. CLS Bank case, 538

Aligning (objects), 277
All the Widgets, lviii, 9, 13, 35, 89, 164, 179, 185, 310, 682
AlphaSlider, 138, 148
Alt (mouse modifier), 93
Alt-Tab, 298
Alternate Reality Kit (ARK), 42, 120, 187, 287, 526
Alto computer, 11, 24, 28, 33, 553
Alto Draw program. *See* Draw application (from PARC)
Alto FTP program (1970s), 158
Ambient displays, 355, 454
Ambient intelligence, 353
AMBIT/G, 51
American Institute of Medical and Biological Engineering (AIMBE), 632
American Sign Language (ASL), 370
Amulet, 14, 57, 289, 492
  command objects, 409–410
  toolkit in C++, 515
Amyotrophic lateral sclerosis (ALS), 424, 426–427
Analogy. *See* Metaphor
Andrew (window system), 25, 44–45, 57, 129–131, 139, 187, 295, 296, 301, 310, 312, 318, 491, 520
Andrew Toolkit, 520
Android, 84–85, 388, 414, 613
  SDKs, 495
  smartphones, 224
  watches, 216
Angle interactor, 515
Angular, 485, 486, 494, 520, 521
Animation, 17, 160–161, 280–281, 370
Apollo, 24
Apollo's Open Dialog tool, 487
App accessibility, 437–439
Apple, 82, 134, 144, 165, 414–415, 456, 490, 531–532, 585, 609–610, 613, 632
  Apple APIs, 611
  Apple II, 28
  Apple iPhone 5S, 638
  Apple iPod (2001). *See* iPod
  Apple Newton, 68–70
    synchronization, 595
  Apple Safari (2003), 47
  Apple Watch, 215–216
  dialog boxes, 250–252
  interaction techniques at, 549–550
  logo, 583
Apple Computer, Inc. v. Microsoft Corp. and Hewlett-Packard Co. (1989), 532, 534, 565
Applets, 47
Appliance, 6. *See also* Consumer Electronics
Application Programming Interface (API), 294, 320, 354, 485, 487, 533, 610
Applicon, 371
Aquamarine, 403
Architecture Machine Group (ArchMac), 11, 36, 55, 316, 333
ARPA. *See* Advanced Research Project Agency
Arrow keys, 95–96, 234, 239, 281
Artificial Intelligence (AI), 441–442, 654
  apocalypse, 458
  models, 467
  systems, 373

Assistive technology (AT), 425, 635
Association for Computing Machinery (ACM), lviii, 7–8, 351, 354, 442, 627
Associative search, 321
Asteroids, 342
AT&T Bell Labs, 56, 298, 491, 637
AT&T Bell telephone company, 205
Atari, 59, 342
Atkinson, Bill, 38, 40, 70, 133, 549, 575–576, 586–587
  General Magic, 584
  HyperCard, 583–584
  initial designs of Lisa, 576–583
Attribute objects, 289
Audio Menu, 167
Augment system, 582
Augmentative and alternative communication (AAC) device, 432
Augmentative Communication, 632
Augmented reality (AR), 55–56, 327, 347, 510, 635, 652
  headsets, 641
  IxTs for, 655
Augmented/virtual/extended reality (AR/VR/XR), 510
  tools for AR/VR/XR interactions, 510
Auto-correction, 217–227
  confounds from, 231
  feature, 218
  on mobile devices, 223–227
Auto-fill, 217–227, 246
Auto-hide (scrollbar), 140
Auto-prediction. *See* Autocomplete
Auto-repeat, 125
Auto-scroll, 120–121
Autocomplete, 217–227, 417
  confounds from, 231
  on mobile devices, 223–227
  on regular computers, 220–223
Autodesk Research, 289, 345, 560
Automated teller machine (ATM), 168
Automatic creation from models, 507–510
Automatic tools, 487
Automotive User Interfaces (Automotive UI), 358
Autosuggest, 222–223
Axure, 487, 506
Azurite, 401, 405

Background spell checking, 219
Backspace, 213, 227, 229, 230, 243, 390, 548
Ballmer, Steve, 599–600
Balloon Help, 415
Balsamiq, 487, 506, 526
Bartram, Lynn, 616
Bat (3D input), 332
Batch document processing system, 233
Baudelaire, Patrick, 34
BBN-LISP, 385
Beat Fitts' law, 115
Beaudouin-Lafon, Michel, lviii, 654
Behaviors, 10, 280–281
Berners-Lee, Tim, 44, 46, 54, 255, 320
Bézier curve, 275
Bin packing, 508
BitBlt, 516
Bitmap, 24, 28, 516
Bitmapped interface, 598
Blackberry. *See* RIM Blackberry Pearl, 100
Blackboard, 552, 634

Blank screen study, 545–546
Blindness, 423, 427–428
Bluetooth, 74, 215, 359, 435, 647
Bob (Microsoft), 320, 454, 604
Body for input/output, using, 638
Booger, 613
Borders of windows, 309
Bot, 438
Bounding box, 265
  size and position for, 271–274
Braille, 428, 439
Brain interfaces. *See* Brain–Computer Interfaces
Brain–Computer Interfaces, 432–433, 655
Bravo, 32, 53, 127, 129, 235, 238, 300–301, 385, 548, 654
  Scrollbar (1974), 127–128
  Text Editor, 32–33
  WYSIWYG text, 385
Brichter, Loren, 2, 609
  pull-to-refresh, 610–614
  short bio of, 609
  spring–action interactions, 614
Bricklin, Dan, 35, 50, 504, 569
  Bricklin's Demo, 50, 487, 504, 572–573
  short bio of, 569
  VisiCalc, 569–572
Bring to Front, 263
Brooks, Fred, 55
Browser (web), 46, 47, 138, 217, 222, 236, 253, 255, 309, 316, 417, 434, 456, 486, 506
Browser wars, 48
Bubble cursor, 115
Burr, Betty, 112
Bush, Vannevar, 53
Business Process Modeling Notation (BPMN), 510
BusMobile, 355
Busy cursor, 158
Button (widget), 167
Buttons on a mouse. *See* Mouse buttons
Buxton, William A. S. (Bill), 12, 57, 149, 472, 475, 559, 655
  adjusting values, 561
  classification, 472–473
  future, 561
  interacting with two hands, 560
  short bio of, 559–560
Byte magazine, 32

C (programming language), 191, 491
C++, 57, 69, 409, 485, 491, 512, 515, 643
C# (programming language), 501
CAD. *See* Computer Aided Design
Calculators, 64, 263
Callahan, John Raymond, 615, 616
Camera angle (in 3D), 329
Camera-based techniques, 336–337
Cancel, 16, 172, 250, 252, 271, 383, 384, 388–390
Cancellable menu, 172
Canvas, 153, 294, 518, 552, 634
Capacitive sensing, 83, 584, 639, 652
CapsLock, 415, 432
Captcha, 438
Car radios, 185
Card, Stu, 112, 473, 476, 621
Card Stunt Program, 544
Carnegie Mellon University (CMU), lvii, 11, 13, 24, 73, 235, 295, 359, 480, 510, 541, 585, 619, 637, 641

Carpal tunnel syndrome. *See* Repetitive stress injury (RSI)
Carr, Robert, 66
Cascading Menus, 165, 172
Cascading Style Sheets (CSS), 47, 235–236, 280, 438, 484, 494–495, 502
Casio FX-7000G, 64
Casio Zoomer, 591–592
Catalyst (for Adobe Flash), 503–504, 515, 526
Cathode ray tube (CRT), 26–27
Cave (virtual environment), 326
Cedar Window Manager, 35, 130, 186, 295, 297, 300, 305, 309, 310, 312, 598–599
Cell phones. *See* Mobile phones
Centipede, 342
Cerebral Palsy (CP), 425–426
Chan, Peggy, 471–472
Character displays, 26
Character recognition method, 536
Character-level error rates, 230
Characters
  for text entry, 193
Characters per second (cps), 125, 228
Chat, 443, 654, 657
Chatbots, 443–444, 447
ChatGPT, 488, 654–655
Chau, Polo, 321
Checkbox, 1–2, 8–9, 15–16, 163–164, 168–169, 175–178, 182, 187–188, 470, 589
CHI. *See* Human–Computer Interaction
Chimera, 400, 403
Chinese text input interaction techniques, 196–200
Choice parameters, 175
Circular magnifying glass, 242
Circular pointing device test, 115
Cirrin, 213
Citrine, 258
Clamshell PDA, 60, 72
Click (mouse button), 93–94
Click-move-click (menu behavior), 182, 616
Click-through prototype, 503–505, 526, 572
Click-to-type, 297
Clickahead, 109
Clicking on objects, 265–266
Clippy, 320, 453–454, 605
Clutching (of a mouse), 97
Cmd window, 50
CMU. *See* Carnegie Mellon University
Code Bubbles, 236
Code completion, 221
Cognitive psychologists, 476
Cognitive walkthrough, 528
CogTool, 479
Collaborative filtering, 455
Color, 278
  blindness, 428
  chooser, 279
  model, 466
  picker, 279
  specifying, 278–279
Combo box, 165–166, 491
Command (mouse modifier), 93
Command history, 384
Command icon, 583
Command Line Interface (CLI), 50, 221, 238, 262, 389

Command object pattern, 408–410, 510
Command-leftarrow, 391
Command-Period (Cmd-dot), 389
Commodore Pet, 28
Common Lisp, 35
Compaq iPAQ, 78–79
Compiler, 219, 475, 484, 584, 598
Components, 10, 57, 488, 500
Computer Corporation of America (CCA), 36
Computer graphics, 55, 275, 325, 510, 544
Computer hardware, 21
  devices, 25–26
  mainframes, 21–24
  personal computers, 25
  personal workstations, 24–25
Computer Science theory, 469
Computer sensing, 332
Computer-Aided Design (CAD), 54, 101, 158, 345, 372
Computer-generated imagery (CGI), 25
Computer-supported cooperative work (CSCW), 56, 245
Computerized UX design tools, 488
Conceptual, 473–476
Cone trees, 473
Confusion matrix, 230, 378
Console window, 28, 50
Constraints, 29, 256, 277, 493–494
Constructive solid geometry (CSG), 345
Consumer electronics, 5, 95, 121, 172, 185, 257, 282, 304, 355, 362, 439
  accessibility, 437–439
  selection in, 282–284
Containers, special kinds of, 495
Context Menu, 145, 165
Context-aware computing, 354
Contextual Inquiry, 480, 525
Contour RollerMouse Red Plus, 653
Control (name for widgets), 9
Control keyboard key. *See* Ctrl
Controlled experiment, 527–530
Conventional GUI interactions, 652
Conversational agent, 443, 456, 458, 654, 656
  UIs for, 654–655
Conway's game of LIFE, 54
Coordinates, 106, 255, 262, 270, 332, 494
  special values for, 494–495
  systems, 270
COPILOT (from 1974), 295
Copilot (GitHub), 222
Copy-and-paste, 2, 244, 258, 262, 289, 387, 544, 546
Copyright, 44, 531–533, 565
CORE, 472
Cornell Program Synthesizer, 237
Corpuz, Simon, 543, 551, 563, 631
Corrected insertions, 230
Corrected no-errors, 230
Corrected omissions, 230
Corrected substitutions, 230
Covered windows, 298. *See also* Overlapping windows
Cox, Norm, 563
  interaction design for scrolling, 566
  legal consulting, 565–566
  mnemonic gestures for interacting with device, 566–567
  short bio of, 563–564
  visual icon design, 564–565

Creating objects
brief history of graphical editing, 284–287
considerations for, 290–291
creating new objects and painting, 264
graphical object hierarchies, 263
research in selection and editing objects, 288–290
selection in consumer electronics, 282–284
selection in spreadsheets, 281–282
selection of graphics, 264–269
selections in linear and hierarchical lists, 282
specifying animations and behaviors, 280–281
specifying colors, 278–279
specifying graphical properties, 279–280
specifying position, size, and shape, 269–278
syntax of operations, 261–263
Cross events, 377
CRT. *See* Cathode ray tube
Crystal, 417–419
Ctrl (mouse modifier), 93
Ctrl-Alt-Del, 389
Ctrl-C, 50, 389
Ctrl-leftarrow, 391
Current window, 296
Cursor, 27, 30, 91–92, 238, 645
Curves, size and position for, 275–276
Custom languages, 512
Cut, copy, paste, 2, 244, 258, 262, 289, 387, 544, 546
Cut-and-paste mechanism, 546–547
Cut-paste-text-method, 479

Damnyouautocorrect.com, 226
Dan Bricklin's Demo Program. *See* Bricklin's Demo
Dance Dance Revolution, 342, 344
DARPA. *See* Defense Advanced Research Project Agency
Dashboard widgets, 9
Data bindings, 494
Data description problem (in PBE), 410
Data detector, 456
Data Glove, 56, 334, 366
De Morgan's law (macro), 410
Deafness, 428–429
DEC. *See* Digital Equipment Corporation
DecWindows, 44, 135
Default value, 176–178
Defense Advanced Research Project Agency (DARPA), 54
Deflecting coils, 26
Degrees of freedom (DOF), 328
DEMO (1985), 504
DENIM system, 506
Dependency analysis, 403
Dertouzos, Michael, 318
Design
decisions, 15
graphic design, 173–176, 191, 526, 601
methods, 480, 526
pattern, 409, 467, 468
prototyping and implementing, 526–527
Designer, 5–6
Desk accessories, 9

Desktop devices, 21
computer hardware, 21–26
display technologies, 26–28
human–computer interaction technology, 48–57
systems, 28–48
Desktop metaphor, 15, 37, 52, 294, 298, 315, 468, 554
alternatives, 315–321
brief history of WM, 294–296
collections of windows, 313–315
considerations for, 321–323
decorations for individual windows, 309–310
icons, 303–306
input focus, 296–298
interaction techniques for windows, 310–313
overlapping windows *vs*. tiled *vs*. tabbed *vs*. single window, 298–303
task bar and dock, 306–309
Desktop publishing software. *See* Word Processor
Desktops, 19, 25, 194–201, 294
Dialog box, 50, 163, 169, 186, 249, 251, 384, 417, 486, 492, 497, 556
design considerations for, 258–260
on mobile devices, 255
research about, 255–258
Dialogs, kinds of, 444–445
Dialogue control, 511
Difficulty of implementation, 379
Digital affinity, 632–634
Digital art board. *See* Graphics tablet
Digital drawing tablet. *See* Graphics tablet
Digital Equipment Corporation (DEC), 22, 24, 44, 296, 389
DEC VT100 display terminal, 27
DigitalDesk, 358
Digitizer, 100
Digitizing, 106
Direct manipulation, 29, 50–52, 458
Direct pointing device, 105–106
Direct selective undo, 403
DirectX APIs, 643
Disability, 423–424, 427, 632
Discount, 528
Discoverability, 364, 419, 461, 527
Disk Operating System (DOS), 28, 40, 50, 125, 234, 303, 434, 572, 598
Disney Research, 637
Display list, 26, 518
Display oriented programmer's assistant, 386
Display PostScript, 44, 518
Displayed menus, 180–181
Displays, 22
list, 26
technologies, 26–28
DLisp, 35
Do What I Mean (DWIM), 218, 456
Dock, 295, 306–309, 322
Document, 294
Document model, 521
Document object model (DOM), 48, 512, 519, 521
Document processors, 236
Documentation, 413, 527
Domain-specific languages (DSLs), 493
Doonesbury, 70
Dorado, 34, 585

DOS (Microsoft). *See* Disk Operating System (DOS)
Double-click (mouse button), 93
Down (mouse button), 93
Drag (mouse), 93–94, 110, 120, 141, 182, 239, 262, 271, 282, 582, 616
Drag-and-drop, 5, 237, 243, 262, 282, 289, 304, 312, 365, 479, 487
Drag-and-Pop, 116, 289
Drag-drop-text-method, 479
Draw application (from PARC), 34, 53, 185, 264, 267, 285
Draw-Text, 516
DrawEllipse, 516
DrawImage, 516
Drawing programs, 33–35, 52–53, 365
Drawing tablet. *See* Graphics tablet
DrawLine, 516–517
DrawRectangle, 516–517
Drop. *See* Drag-and-drop
Drop-down list. *See* Combo box
Drop-Down Menu. *See* Menu Bar
Druin, Allison, 339
Dubberly, Hugh, 452
Dvorak keyboard layout, 195
DWIM. *See* Do What I Mean
Dynabook, 24, 51, 586

E-forms, 318
Eager recognition, 367
Early smartphones, 71–73
Early wireless phones, 70–71, 188
Eclipse IDE, 220
Edge (Browser). *See* Microsoft Edge
EdgeWrite, 204, 223, 369, 378, 432
Edit points, 275
Editing
  gestures for, 370–372
  operations for, 243–245
  research in editing objects, 288–290
  systems for code, 236–237
  text. *See* Text editing
Effective text entry requirements, 628
Effectiveness, 91, 157, 231, 364, 527
Efficiency, 6, 379, 628
Efficient behavior, 628
Electroencephalograph (EEG), 432–433
Electronic programming guide, 283
Elevator (scrollbar). *See* Thumb
ELIZA program, 443
Emacs (Text editor), 53, 234, 236, 238, 240, 295, 300–301, 391, 394–397, 400, 410
Embodied intelligent agents, 452–455
End-user development (EUD), 500
End-user programming (EUP), 40, 500
Endless scrolling. *See* Infinite scrolling
Engelbart, Doug, 30, 52, 53, 56, 96, 201, 295, 549
English, Bill, 96, 112
ENIAC, 22
Ergonomics, 7, 91, 113, 424, 476, 528
Ericsson, 73–74
Error-proneness, 228, 527
Errors, 6
Esc key, 121, 154, 172, 238, 271, 311, 383, 470
European Particle Physics Laboratory (CERN), 46

Evaluation, methods for, 527
controlled experiments, 528–530
usability evaluation, 527–528
Evans&Sutherland, 55
Event handlers, 512
Event model, 110, 246, 367, 466, 511–513
Everything Delete Insert T (EDIT), 238
Exertion games, 344
ExperTelligence, 499
Explainable AI, 460
Extended reality (XR), 328, 510

Fabrik, 179, 252
Facebook's app, 612
Fake news, 460
Falcon (3D haptic device), 338
False negative gestures, 378
False positive gestures, 378
"Fast" (Palm Pilot principle), 593
Feature phones, 63–64, 70
Feedback, 267, 270–271, 337, 447
Feldspar, 321
Fields (of a form), 50, 218, 246, 255. *See also* Text entry fields
Figma, 315, 487, 506, 526
Final feedback, 271
FingerSense (touchscreen technology), 641–642
Finite State Automata. *See* State Transition Diagrams
Finite-state machine (FSM). *See* State Transition Diagrams
First-one-only menu behavior, 182
First-touch, 93
Fitbit, 361
Fitts' Law, 111–115, 151, 476, 620, 644, 654
Fitts, Paul M. Jr., 111
Fixed Menu, 165, 170
Flash Catalyst. *See* Catalyst
Flash system, 47, 357, 434, 485, 494, 503
Fledermaus, 332
Flex (toolkit), 494
FlexBox, 495
Flick gesture, 144, 366, 377
Flick-to-scroll, 67, 83, 144, 365, 372
Flip-if-change-sides, 272
Flying hands, 646
Fold-and-drop, 289
Foley, Jim, 471–472, 475, 514
Force feedback, 56, 337
Force-sensitive joystick, 99
Form, 50, 163, 249
Form Editors. *See* Interface Builder
Form fill-in, 50, 218
Format painter, 262
Formatted text, 46, 233, 245, 415
FORTRAN, 472
Fossil Company, 76–77, 215
Fractals, 346
Frame, 294
FrameMaker, 45, 236
Framework, 486, 489–492, 613
Frankston, Bob, 35, 569
Freehand curve, 276
Freehand scribbles, size and position for, 275–276
Freeze Panes, 571
French, Melinda, 604
FRESS, 55
Furness, Tom, 55

Gadgets, 10
Gain function (of pointing device), 97, 106, 577, 620–621
Galaxy, 496
Games, IxTs for, 4, 54, 325, 341–344
Gargoyle, 278, 286
Garnet, 14, 35, 57, 289, 492–493
Gates, Bill, 81–82, 458, 600–603
Gaussian distribution formula, 629
Gaze tracking, 104, 153, 429
GDPR (General Data Protection Regulation), 460
Gelernter, David, 317
General Magic, 70, 584
  "Magic Cap" OS, 70
General-purpose constraints, 494
Generations (of wireless networks), 73
Generative AI. *See* Conversational Agent
Generative Pre-trained Transformer (GPT), 222
Geometry Management, 492
Geschke, Chuck, 35
Gestural alphabets, 203–205, 366
Gestural text entry, 205
Gesture, 63, 144, 146, 179, 203, 365–366
  advantages of, 366–367
  applications for, 368–369
  for commands, 372–373
  considerations for gesture recognition, 379–381
  disadvantages of, 366–368
  evaluating and improving learnability and accuracy of, 378–379
  flick-to-scroll, 144
  funny, 379
  interactor, 515
  for object entry and editing, 371–372
  one-and two-finger scrolling, 145–146
  for proofreading and editing text, 370–371
  recognition algorithms, 373–377
  recognizing sign language gestures, 370
  for scrolling, 144
  for text entry, 369–370
Gesture recognition, 55, 63, 328, 365, 458
  algorithm, 365
  software, 365
Gilt (interface builder), 255
Gina, 409
Giraffe game, 343
Glass (Google), 326, 327, 372
Gloves, early tracking of, 333–335
Go Corporation, 65, 66, 70, 188, 369
Goals, operators, methods, and selection rules (GOMS), 476, 478–480
Goldberg, Adele, 586
Goldstein, Irwin, 22
GOMS Model. *See* Goals, operators, methods, and selection rules (GOMS)
Google, 84, 140, 221–223, 454, 460, 520, 533, 627, 637
Google Chrome (browser), 47, 138–139, 223, 304, 309
Google Docs, 220, 245, 297, 406
Google Pixel 2 Phone, 340, 445, 630
Google Suggest, 222
Google Wave, 136, 566

Gorilla arm syndrome, 95, 116, 349
Graffiti, 76, 204, 378, 592
  gestural language, 592
Granularity, 244
Graphic design, 173–176, 191, 526, 601–602
Graphic designer, lxi, 526
Graphical control element. *See* Widget
Graphical editing, brief history of, 284–287
Graphical editors, 264, 289, 392
Graphical objects, 7, 50, 146, 261, 284, 349, 372, 407, 471
  hierarchies, 263
Graphical properties, 279–280
Graphical search, 288
Graphical user interface (GUI), 7, 575, 651
  builder tools, 483, 499
  IxTs, 656
Graphical widget, 10
Graphics, 518
  selection of, 264
  tablet, 100–102. *See also* Tablet
Graphics Language (GL), 510
Graphics tablet, 100
Graspable user interfaces. *See* Tangible User Interfaces
Grid, 274, 276, 291, 495
GridBagLayout, 495
Gridding, 276
GRiDPad, 67
Group (of graphics), 263, 265
Guessability, 204, 379
GUI. *See* Graphical user interface
GUI Builders. *See* Interface Builder
Guiard's Kinematic chain model of bimanual asymmetric motor action, 560
Guitar Hero, 343
Gypsy (text editor), 548

Haitani, Rob, 591
  Casio Zoomer, 591–592
  Palm Pilot, 592–595
  short bio of, 591
Halftone, 28, 298, 516
Hamburger icon, 168, 250, 563–564
Hand printing, 202–203
Hand printing recognition, 63, 368
Hand scrolling, 141, 346
Handheld computers, 60
Handheld devices, 21, 60–61, 307, 340, 361, 533, 595
Handheld PC, 60–61, 72, 78, 533
Handheld projector, 336, 361–362
Handles (curvature), 276
Handles (selection), 242, 266–268, 275, 289, 331, 349–350, 644–645
Handspring, 76
Handwriting, 63, 81, 196, 199, 201–203, 365, 369, 448, 652
Handwriting recognition, 63, 68–70, 81–82, 203, 366, 368, 373, 592
Haptic feedback, 337, 624, 644
  input techniques with, 337–339
Harrison, Chris, 637
  using body for input/output, 638
  context, 638
  increasing richness of touch interfaces, 639
  short bio of, 637–638

Hawking, Stephen, 426, 432, 458
Hawkins, Jeff, 67, 74, 76, 204, 592
HCI. *See* Human–Computer Interaction
Head tracker, 327, 429
Head-crusher technique, 348
Head-mounted display (HMD), 56, 121, 326, 355, 361, 372
Hearing loss, 428–429
Help modes, 413–414, 417
  autocomplete as, 417
  considerations for, 421
  screen item, 417–419
  status bar, 414–415
  stencils, 420–421
  tooltips, 415–417
Herot, Chris, 36
Heuristic evaluation (HE), 528
Hewlett Packard (HP), 24, 64, 78, 263, 532, 534, 565, 586
Hierarchical command objects, 409
Hierarchical lists, selections in, 282
Hierarchical Menu, 165
Hierarchical Task Analysis, 480
Hierarchical view, 282
High-end camera-based systems, 336
High-fidelity prototypes (hi-fi prototypes), 503
Historical menus, examples of, 185–188
History Brush (Photoshop), 400
Hoaloha Robotics (company), 607
Hold (mouse button), 93–94
Holographic Remoting Player, 647
HoloLens, 327, 642
HoloLens 1 (augmented reality headset), 642, 646
HoloLens 2
  interaction model, 642–643
  prototypes and interactions, 643–646
Home Page Reader (HPR), 434
Hopkins, Don, 179, 615
  definitions and key properties of pie menus, 616
  history of pie menus, 616–617
  short bio of, 615
Horn, Bruce, 585
  Apple Macintosh, 587–589
  PARC and Smalltalk, 586–587
  short bio of, 585–586
Hotkeys. *See* Keyboard shortcuts
HotSync, 595
Hover state, 1–2, 102, 106–107, 182, 280, 470
Hoverbox, 415
Hovered, 181
HP-65 calculator, 64
Hsu, Jason, 543, 551, 563, 631
HTML. *See* HyperText Markup Language
HTML+, 255
Hullot, Jean-Marie, 57, 499, 541
Human factors, 7, 100, 206, 528
Human Performance Models, 467, 476
  goals, operators, methods, and selection rules, 478–480
  keystroke-level model, 476–478
Human–Computer Interaction (HCI), 7–8, 13, 48, 525
  interaction and application styles, 50–56
  technology, 48
Human–Computer Interaction Lab (HCIL), 147
Human-device input methods, 627

HyperCard, 40, 43, 54, 75, 183, 187, 259, 303, 468, 487, 505, 516, 575, 583, 593
HyperTalk, 40, 505, 512, 583
Hypertext, 46–48, 53–54, 321, 413, 584
Hypertext Editing System, 53–54
HyperText Markup Language (HTML), 46, 48, 157, 233, 279, 417, 438, 484, 486, 494–495
Hypertext Transfer Protocol (HTTP), 47
Hyperties, 54

IBM, 24, 600, 619, 625, 627, 629
IBM alphaWorks, 628
IBM OS/2 Presentation Manager, 295, 311, 600–602
IBM PCs (1981), 40, 599
IBM Research, 637
IBM TrackPoint. *See* TrackPoint
Icon, 253, 294, 302–306, 552–553
  alternatives, 315–321
  brief history of WM, 294–296
  collections of windows, 313–315
  considerations for, 321–323
  decorations for individual windows, 309–310
  input focus, 296–298
  interaction techniques for windows, 310–313
  overlapping windows *vs*. tiled *vs*. tabbed *vs*. single window, 298–303
  task bar and dock, 306–309
Iconify (window), 312
IDE. *See* Integrated development environment
IEEE 802.11, 73
Igarashi, Takeo, 347
Illustrator (Adobe), 35, 53, 142, 261, 264, 274, 506, 521
Imaging model, 294, 466, 515
  retained object model, 518–519
Implementation, models and patterns for, 510–521
Implementer, 5
Inclusive design, 425
Incremental search, 223
Index of difficulty (Fitts' Law), 113
Indirect pointing device, 105–106
Individual user interface generators (IUIGs), 636
Individual windows, decorations for, 309–310
Industrial Revolution, 628
Infinite scrolling, 140–141
Infix, 64, 262–264, 476, 571
Info-bot, 636
Infopop (Eclipse tooltip), 416
Information appliance, 353–354
Information in scrollbars, 138–140
Information Technology Center (ITC), 57, 295
Infrared signals (IR signals), 65
Ingalls, Dan, 549, 586
Inking, 34, 264, 276, 365, 515
Input (to computer), 7
  using body for, 638
  with haptic feedback, 337–339
Input box. *See* Text entry field
Input devices, 91, 109, 116, 296, 332–340, 429, 471–473
  for 3D IxTs, 332–340
  Buxton's classification, 472–473
  Card, Mackinlay, Robertson Model, 473

Foley and Wallace and Chan, 471–472
models, 471
Input field. *See* Text entry fields
Input focus, 110, 128, 246, 260, 296–298, 311, 322, 513
Input model, 486
Input technique. *See* Interaction technique
Insert mode, 233, 238, 415, 432, 548
Instinctual interaction, 643
Integrated development environment (IDE), 219, 221, 236, 416, 484, 544
Intellectual property, lxii, 14, 531
considerations for, 538–539
Intelligent agents, 441, 452–455
Intelligent interaction technique, 457
Intelligent interface, 56, 442, 458
algorithms and machine learning for content selection, 455–456
challenges, risks, and controversy about, 458–461
definitions, 442–443
embodied intelligent agents, 452–455
kinds of dialogs, 444–445
multi-modal interaction techniques, 450–452
speech and natural language interfaces, 443–450
users provide input, 445–447
IntelliMouse, 121, 146
IntelliSense, 221
Interaction Museum, 654
Interaction Technique (definition), 8–9
Interactive GUI tools, 487, 499–503
Interactive programming guide. *See* Electronic programming guide
Interactive prototyping tools, 487, 503
examples of prototyping tools, 504–506
kinds of prototyping tools, 503–504
research on prototyping tools, 506–507
Interactive visualizations, 11
Interactors model, 492, 514–515
Interface Builder (IB), 57, 487, 499, 506, 541, 573
Interface Development Tools (IDT), 499
Interim feedback, 270–271
Interim-selected, 181
Interleaf, 45, 236
Interlisp, 35, 295
Interlisp-D, 35, 129–131
Internationalization, 18, 496–498
Internet companies, 460
Internet Explorer, 47, 604
Internet of Things (IoT), 355, 362–363
InterPress, 35
InterState, 470, 502–503
InterViews, 256, 492, 496
Intrinsics (of a Toolkit), 490
Invalid items, 171, 181, 259, 493
Invention process, 553
Inverse model (for Undo), 403–405
iOS, 40, 59, 93, 175, 212, 224, 242, 261, 388, 437, 445, 485, 519, 609
iOS Spotlight, 321

iPad, 85–87, 634
iPhone, 83–84, 145, 188, 224, 242, 305, 308–309, 609, 629, 634
  3D touch, 109
  scrollbars, 138
  spin menu, 246
iPod, 82, 121, 188, 526, 653
Ishii, Hiroshi, 354, 359
ISO9241-9 standard, 115
Isometric joystick, 99
ITC. *See* Information Technology Center
ITS (system from IBM), 257, 508
iTunes store, 82
IxT, 9

Jacquard by Google. *See* Levi's Jacket with Jacquard by Google
Jade (UI layout tool), 257, 508
Jasper, 236
Java "GridBag Layout", 495
Java language, 47, 495
JavaScript, 29, 48
Jazz, 316–317
Jitter, 102
Job Access with Speech (JAWS), 434
Jobs, Steve, 38, 42, 68, 134, 295, 499, 517, 549–550, 586–587
John, Bonnie, 479
Johnson, Timothy, 55
Jot (text entry method), 80, 204, 369
Joystick, 36, 54, 97–100, 105, 112, 143, 208, 316, 332, 342, 429

Kaehler, Ted, 169, 186, 586, 589
Kanerva, Pentti, 547
Kare, Susan, 141, 583, 585, 588
Kasik, David, 56
Kay, Alan, 24, 31, 51, 52, 295, 388, 552, 586
Keyboard, 1, 15, 30, 61, 72, 80–81, 104, 109–110, 125, 167–168, 183–185, 194–201, 208–215, 250, 296–298, 343, 387, 429, 446, 478, 561, 620, 628
  operation, 183–185
  shortcuts, 183
Keyboard focus, 181. *See also* Input focus
Keypad, 30, 63, 71, 167, 201, 205–208
Keys, 125–126, 167
Keystroke-Level Model (KLM), 476–479
Keystrokes per character (KSPC), 207, 228
Kim, Jae, 543, 551, 563, 631
Kinect, 105, 336, 348, 350, 370
Knob, 29, 95, 121–123, 316, 332, 359, 472, 653
Knots, 275
Knowledge Navigator, 452
Knuth, Donald, 235, 496
Krueger, Myron, 55
Kurlander, David, 288
Kyocera QCP-6035, 76

LabVIEW, 357, 417
Lamport, Leslie, 235
Lampson, Butler, 32
Landmark events, 317
Language (for text formatting), 233, 234
Language models, 467, 629
Lanier, Jaron, 334
Laptops, 19, 25, 60
Large Language Models (LLMs). *See* Conversational Agent
Laser pointer, 105, 111
Laser printing, 31, 235

Lasso, 269, 286, 576
LaTeX, 235
Layers (in graphics), 266, 401
Layout management, 178–180, 492
  absolute layout, 492–493
  general constraints, 493–494
  special kinds of containers, 495
  special values for coordinates, 494–495
  stretchy spacers, 496
Layout Managers, 492, 494
Le Lisp system, 499
Leap Motion, 337, 652
  device, 652
  sensors, 370
Learnability, 6, 378, 453, 527
  of gestures, 378–379
Learning, 6, 228–229
  time, 228
Learning curve (for text entry), 228
Left-down (mouse button), 93
Legal consulting, 565–566
Legal issues
  considerations for intellectual property, 538–539
  copyrights, 532–533
  patents, 534–538
  trade secrets, 534
  trademarks, 533
Lego Company, 356
Levi's Jacket with Jacquard by Google, 361
Lexical, 475–476
Lifestreams, 317
Light Handles, 51
Light-emitting diodes (LEDs), 638
Lightpen, 94–95, 349
Lincoln Wand, 55
Line editors, 233
Linear lists, selections in, 282
Linear menus, 616
Linear model, 394–395
Linear transfer function, 621
Lines, size and position for, 275–276
Linux, 617
Lisa (Apple), 38, 52, 133, 165, 171, 176, 179, 183, 187, 238, 249, 250, 254, 264, 267, 271, 274, 295, 298, 304, 309, 310, 386, 468, 575
  initial designs of, 576–583
  scrollbars, 133–135
Lisp, 24, 35, 57, 130, 385, 499, 508, 514
Lisp Machines Inc. (LMI), 24, 295–296
Listener Pattern, 296, 513–514
Listener window. *See* Input focus
ListPad, 227
Localization, 18
Logical hierarchy, 190
LOGO, 355
Long-press, 3, 110, 165, 243, 365, 415
Look and feel, 10, 17, 191, 257, 490, 508
Lou Gehrig's disease, 426
Loupe. *See* magnifying glass
Low-fidelity prototypes (lo-fi prototypes), 503
Low-pass filter, 105
Lucasfilm, 25
Lyu, Tianying, 543, 551, 563, 631

MacApp, 409, 490, 521
MacDraw, 53, 264
Machine learning, 222, 373, 411, 455
  for content selection, 455–456
  systems, 456

Macintosh, 133, 221, 235, 262, 286, 295, 321, 386, 572, 575, 585, 587, 597
  Human Interface Guidelines, 183
  scrollbars, 133–135
Macintosh Style Guidelines, 490
Mackay, Wendy, 654
Mackinlay, Jock, 473
MacPaint, 52–53, 141, 264, 269, 286, 575, 593
MacPaint 1.0, 141
Macro, 385, 410
Macro recording and programming by example, 410–411
Macromedia, 47, 356, 494, 505, 510, 512
Macromedia Director, 505, 510
Macromedia Flash. *See* Flash system
MacTerminal program, 159
Maes, Pattie, 458
Magic Cap, 70
Magic lens. *See* Toolglass
Magnifying glass, 242
Mailbox (email app), 614
Mainframes, 21–24
Man-Machine interfaces. *See* Human–Computer Interaction (HCI)
Manipulating objects
  brief history of graphical editing, 284–287
  considerations for, 290–291
  creating new objects and painting, 264
  graphical object hierarchies, 263
  research in selection and editing objects, 288–290
  selection in consumer electronics, 282–284
  selection in spreadsheets, 281–282
  selection of graphics, 264–269
  selections in linear and hierarchical lists, 282
  specifying animations and behaviors, 280–281
  specifying colors, 278–279
  specifying graphical properties, 279–280
  specifying position, size, and shape, 269–278
  syntax of operations, 261–263
Marching ants effect, 269, 575
Marking menus, 167, 179–180
Markup (of text), 233
Markup application (from PARC), 33, 53, 127, 185, 264
Mary, Virgin, 552
Massachusetts Institute of Technology (MIT), 11, 23, 28, 36, 45, 52–56, 218, 234, 295, 333, 356, 385
Maximize (window), 311
Maya, 289, 350
McCarthy, John, 24, 623
McGregor, Scott, 295, 598, 602
MEMEX, 53
Memorability, 6, 367, 379
Menu bar, 132, 165, 170, 173, 176, 178, 180, 187, 253, 297, 491, 575
Menu Button, 168, 186, 250
Menu interactor, 515
MenuLay system, 57, 506
Menus, 17, 163, 576
  behaviors, 181–185

categories, 164–169
contents, 188–190
design considerations, 190–191
dimensions, 169–185
examples of historical menus, 185–188
Mesa (programming environment), 35
MessagePad 2000, 592
Metaphor, 467–469
Metaphor Computer Systems, 566
Metropolis Virtual Keyboard, 212
Mickey, 256
Micro-gestures, 645
Micro-interaction, 10
Micro-targeting, 460
Microelectrodes, 433
Microphone, 357, 446
Microsoft, 81, 201, 531, 565, 597, 600, 625, 637, 642, 646
Microsoft Basic, 598
Microsoft Bob, 320, 604–605
Microsoft DOS applications, 28, 598
Microsoft Edge (browser), 47, 634
Microsoft Flight Simulator, 598
Microsoft HP6-00001 Surface Hub, 109
Microsoft Internet Explorer (browser), 47, 604
Microsoft Kinect, 105
Microsoft Office XP, 188
Microsoft Paint, 52, 261
Microsoft Pascal, 598
Microsoft Pen Windows, 68
Microsoft PowerPoint, 5, 261, 506
Microsoft products, 605–607
Microsoft Research, 81, 637
Microsoft User Interface Design Group, 602–603
Microsoft Windows, 40, 187, 295, 297, 312, 387, 598–601, 644
precursors to, 598
system, 252
Microsoft Windows 1.0, 187, 295, 598–601
Microsoft Windows 2.0, 187, 598–601
Microsoft Windows 3.0, 598–601
Microsoft Windows 7, 312
Microsoft Word, 10, 164, 166, 218, 232, 235, 240, 252, 297, 305, 310, 390, 408, 414, 497, 555, 634
Midas problem, 153
Millennium, 605
Mindstorms, 356
Minicomputers, 22
Minimize (window), 312
Minority Report (movie), 336
Misrecognition, 368
MIT. *See* Massachusetts Institute of Technology
MIT Media Lab, 11, 36, 316, 336, 354, 359, 619
Mixed reality (MR), 328
Mixed Reality Toolkit (MRTK), 646–647
Mixing console, 124
MMI (Man-Machine Interfaces). *See* Human–Computer Interaction (HCI)
MMM, 109, 289
Mnemonic (for menu items), 184
Mnemonic gestures for interacting with device, 566–567
Mobile device, 59–60, 211
Android, 84–85
Apple Newton, 68–70

auto-correction and autocompletion on, 223–227
calculators, 64
dialog boxes on, 255
early smartphones, 71–73
early wireless phones, 70–71
General Magic's "Magic Cap" OS, 70
gestures, 63
GRiDPad, 67
and handheld devices, 60–61
handwriting and hand printing recognition, 63
iPad, 85–87
iPhone, 83–84
iPod, 82
Microsoft Pen Windows, 68
mobile phone generations, 73
mobile phones, feature phones, and smartphones, 63–64
Palm, 74–77
palm-size devices, tablets, and wearables, 61–62
PDA, 63
pen computing, 62
pen operating systems, 65–67
positioning text cursor and selecting text on, 241–243
RIM Blackberry, 80
scrollbars, 137
text entry for, 201–217
touchscreens and touchpads, 61
ubiquitous computing, 64–65
Windows CE, PocketPC, 78–80
Windows TabletPC, 81
wireless networks, 73–74
Mobile phone generations, 73
Mobile phones, 48, 63–64
Mode, 71, 92, 166, 238, 264, 410, 417, 547–549, 581
Moded editing, 238
Model, 466–467
model-based system, 507
Model Human Processor, 476
Model-based design, 466, 487, 510
Model-view-controller (MVC), 466, 519–520
document model, 521
pattern, 486
Modeless editing, 238
Modeless software, 547–548
Moggridge, Bill, 591
Monetary Value of Interaction Techniques, 14, 531
Monochrome (displays), 28, 516
Moran, Tom, 476
Morphic, 634–635
MorphicBar, 634
Morphology, 218
Mosaic, 46, 47
Mother of All Demos, 30
Motif, 45, 135, 296, 490
Motif "Form", 495
Motion capture, 336
Motorola Razr, 71
Mouse, 96, 545, 578, 599
buttons, 93–94
cursor, 92
IxTs using, 344–347
Mouse buttons, 107
Mouse cursor. *See* Pointer cursor
Mouse Keys, 429
Move-grow interactor, 515
Move-to-type, 297
Moving menu, 186, 362
MS-DOS. *See* Disk Operating System (DOS)

Multi Player SimCityNet game, 617
Multi-device Multi-user Multi-editor system (MMM system), 109, 289
Multi-Level Undo, 384, 394–398
Multi-modal interaction techniques, 450–452
Multi-modal user interfaces, 450, 655
Multi-tap (text entry), 71, 206–207
Multi-user text editing, 245
Multi-user undo, 406–408
Multimedia, 55, 505
Multiple desktops, 313, 323
Multiple Document Interface (MDI), 299
Multiscale interface. *See* Zoomable User Interface
Muscular Dystrophy (MD), 427, 429
Muscular problems, 425–427
Music system, 561
Mutual disambiguation (in speech recognition), 448

N-gram, 629
Namco, 342
Names (for Interaction Techniques), 9–10
Naming test, 564
Nass, Cliff, 454
National Center for Supercomputer Applications (NCSA), 46
National Institute of Mental Health, 38
National Science Foundation (NSF) (US), 28
Natural language, 442
  dialog systems, 444
  feedback, verifying, and correcting, 447–449
  interfaces, 443–450
  kinds of dialogs, 444–445
  and speech, 56
  system actions, 449–450
  users provide input, 445–447
Natural language generation (NLG), 443
Natural language processing (NLP), 442
Natural language understanding (NLU), 442
Naturalness, 378–379
Neatening, 372
Negroponte, Nicholas, 36
Nelson, Ted, 53
Netscape Navigator (browser), 47–48
Network extensible Window System (NeWS), 44–45, 263, 309, 518, 616
New Points interactor, 515
Newell, Allen, 476
Newman, William, 51, 57
NeWS. *See* Network extensible Window System (NeWS)
Newton MessagePad, 68, 203, 211, 369, 591
NewtonScript, 68
NeXT, 17, 40, 42, 44, 57, 134, 136, 182, 183, 187, 295, 307, 389, 485, 487, 491, 496, 499, 517, 518, 526
  Interface Builder, 44, 499
  scrollbars, 135–137
Next next generation interfaces (NNGIs), 635
Nielsen, Jakob, 171
Nintendo Wii. *See* Wii

Nintendo Wii controller, 105, 209
NLS, 30, 52, 56, 201, 233, 238, 245, 295, 546
NLS Journal, 53
Nodes, 469
Nokia, 70
Non-dominant hand, 560
Non-interactive hologram objects, 645
Non-OO programming languages, 491
Non-Practicing Entity (NPE), 538
Non-recognition substitutions, 230
NOOBIE, 339
Norman, Don, 6, 48, 354, 388, 466
Notch, 121, 123
Novint, 338
Nroff, 235
NSF. *See* National Science Foundation
Nuance Communications, 214
Nudging (graphical objects), 274, 390
NumLock, 432
NXT, 356
NXT-G, 357

Object entry, gestures for, 371–372
Object model, 466
Object Pascal, 491
Object-oriented classes (OO classes), 486, 491
Object-oriented drawing system, 289
Objective-C, 491
Observer Pattern, 514
Oculus camera-based hand tracking, 652
  of pointing, 110
Office Assistant. *See* Clippy
Office of Naval Research (ONR), 54
Ohlfs, Keith, 499
OK and Cancel, 177, 252
oN-Line System. *See* NLS
On-screen displays (OSD), 187
On-screen keyboard, 209. *See also* virtual keyboard
On-screen menu, 168
On-screen slider, 146
One-dimensional (1D) input, 119
  devices for 3D, 91, 332
  physical one-dimensional sliders, 123–124
One-finger scrolling, 145–146
ONR. *See* Office of Naval Research
Open Data Model, 318–320, 466, 656
OpenLook, 44–45, 135–137, 183, 187, 259, 296, 490
Operating system, 295, 485
OPTI virtual keyboard, 212
Option menu. *See* Combo box
Option sheets. *See* Dialog box
OPUS editor, 501
Oracle America, Inc. v. Google, Inc., 533
Original position, 240
OS/2. *See* IBM OS/2 Presentation Manager
Oscilloscope, 26
Outline view. *See* Hierarchical view
Output (from computer), 7
  using body for, 638
Over-trust, 460
Overlapping windows, 298–303
Overshoot (for pointing), 110
Overshoot (for scrolling), 121, 126, 136, 143, 152

Pac-Man, 342
PAD, 316–317
PAD++, 316–317
Paging (for scrolling), 126
Painting programs, 33–35, 52–53, 261, 264, 518–519
  selecting pixels for, 268–269
Palette, 166, 262
Palm, 74–78, 121, 125, 173, 204, 211, 255, 305, 309, 343, 369, 429, 484, 531, 533, 591–595, 628
  devices, 188
  palm-size devices, 61–62
  scrollbar, 137
  up/down arrows, 125
  watch, 305
Palm Computing, 74
Palm PC, 533
Palm Pilot. *See* Palm
Palm UI, 76
Palm-size device, 61
Pane, 294, 571
Paned windows, 299–300. *See also* Tiled windows
Panning, 119, 315
Paper prototypes, 504
Paper tape, 23
Parallax, 102
PARC. *See* Xerox Palo Alto Research Center (PARC)
PARC tab, 65, 369
Parkinson's disease, 427
Participatory design, 378
Particle systems, 346
Pascal (non-OO programming language), 491
Patent Assertion Entity (PAE). *See* Non-Practicing Entity (NPE)
Patents, 3, 14, 76, 144, 194, 203, 204, 207, 208, 214, 253, 373, 415, 417, 531, 534
  controversies, 537–538
Pausch, Randy, 334
PC. *See* Personal Computer
PDA. *See* Personal Digital Assistant
Pebbles Shortcutter, 121
PebblesDraw, 109, 289, 407
Pen computing, 62
Pen model, 518
Pen operating systems, 65–67
Pen Windows, 68, 369
Pencept, 65, 369
Pending delete, 218, 239, 241, 246
PenPad, 65, 369
PenPoint, 65–66, 202, 241, 369, 370, 468
  operating system, 369
People with disabilities, 423
  adaptations, 435
  concepts and definitions, 425
  consideration, 439
  interaction techniques, 429–433
  kinds and causes, 425–429
  screen readers, 433–435
  web, app, and consumer electronics accessibility, 437–439
Percent-done progress indicators, 12, 157
  design considerations, 162
  history, 158–160
  implementation, 160
  variations, 160–161
Peridot, 14, 35, 457, 460, 502
Perkins, Rod, 581
PERQ, 11, 24, 159, 516

PERQ Systems. *See* Three Rivers Computer Corporation
Personal computer (PCs), 21, 25, 40
Personal Digital Assistant (PDA), 21, 63, 68, 137, 303, 509
Personal Dynabook. *See* Dynabook
Personal Dynamic Media, 586
Personal Universal Controller (PUC), 257, 509
Personal workstation, 24–25, 44
Personalized menus, 188
Personally consistent user interfaces, 509, 636
Perspective wall, 473
Pervasive computing, 353
PHANTOM (6D haptic device), 338, 347
Phidgets, 354–355, 360
PHIGS, 472
Phone Tree. *See* Audio Menu
Photoshop (Adobe), 52, 53, 142, 261, 318, 393, 395, 400, 411, 506, 521, 526
Physical buttons, 125–126
Physical interaction techniques, 4
Physical inventions, 534
Physical knobs, 121
Physical menus, 185
Physical one-dimensional sliders, 123–124
Physical Widgets, 354–355, 359–360. *See also* Phidgets
Pick-and-Drop, 289
Pico projectors, 361
Pie menus, 167, 179–180, 615
  definitions and key properties of, 616
  history of, 616–617
  timeline, 616
Piezoceramic benders, 334
Pilot (Palm Pilot). *See* Palm
Pinch-to-zoom, 83, 146, 645
Pinned menus, 183
Pinyin system, 197, 199
Pitch, 329
Pixel, 28
  Google Pixel phone, 340, 445, 627
  phone squeeze interactions, 627
  squeeze-to-assist in Google Pixel 2 Phone, 630
PIXIE system, 179
pixmap, 28
Plain text, 232
Plaintiff, 537
Plaisant, Catherine, 147
Play position for video players, 148
Plugins, 656
POBox (autocomplete), 223
PocketPC, 78–80
Point (size of fonts), 270
Pointer, 30
  operation, 182–183
Pointer cursor, 91–92
Pointing device, 7, 91, 528
  arrow keys, 95–96
  considerations, 116–117
  cursor and selection, 91–92
  dimensions for, 105–109
  graphics tablet, 100–102
  IBM TrackPoint, 100
  issues, 109–110
  joysticks and trackballs, 97–100
  lightpen, 94–95
  mouse, 96–97
  mouse buttons, 93–94
  remote interaction, 104–105
  research, 115–116

speed and accuracy testing, 110–115
touchscreens, 102–104
Pointing Stick. *See* TrackPoint
Pokémon Go game, 327
Polhemus, 332
Polylines, size and position for, 275–276
Pong game, 342
Pop-Up Menu (on the Macintosh). *See* Combo box
Popup Menu, 164, 170
Portable chording keyboard, 208–209
Portable device, 61
Pose, 328, 366
Position controlled joystick, 99
Positioning text cursor
on desktops, 238–241
on mobile devices, 241–243
Postfix, 262, 284
PostScript, 35, 518
graphics language, 309
Power Glove, 334
PowerPoint, 52, 261, 263, 264, 272, 390, 506, 521
Pragmatic level model, 473–476
Predictive-text, 207
Prefix, 50, 262, 264
"Preflighting" concept, 588
Prentke Romich Company, 432
Presentation Manager. *See* IBM OS/2 Presentation Manager
Press (mouse button), 93–94
Press-and-drag, 239
Press-and-hold, 93
Press-drag-release behavior, 182
Pretty-printing, 236
Procedural generation, 346
Product model, 467
Productivity, 6, 14, 24
Programmable calculators, 64
Programming by demonstration (PBD), 280, 410, 457
Programming by example (PBE), 280, 358, 410, 448, 457, 502, 552, 553
Programming language model, 518, 598
Progress bars, 12, 157
Project Athena, 296
Projector, 336, 361
PROMIS, 54
Prompt engineering, 655
Proofreading, 55, 370
Property, 531
Property Sheets, 38, 186, 249, 499, 556
Prototype-instance object model, 29, 42, 69, 466
Prototypers, 503
Prototyping and implementing, 526–527
Prototyping tools, 503
Pull-to-refresh, 2, 609–613
interaction technique, 541, 609
patent, 614
Punched cards, 22–23
Push-to-talk, 445
Put-That-There, 333, 444, 450
Pygmalion, 303, 553

Qeexo, 639, 641
Quadruple clicks, 93
Qualcomm, 637
Quality of UI, 6
Quest (Head-mounted display), 327, 646

Questionnaire for User Interface Satisfaction (QUIS), 528
Quick fixes (in an IDE), 220
QuickDraw, 575
Quickset, 448, 451
QuickType, 226
Quikwriting, 213
Quill system, 378
QWERTY, 15, 194, 199, 209, 211, 212, 247, 387, 628

Radial menus. *See* Pie menus
Radio button, 38, 47, 163–164, 168–169, 174, 177, 185–190, 255, 507, 589
Radio frequency (RF), 63
RAND tablet, 54, 101, 201, 369
Randomized controlled trials (RCTs), 527. *See also* Controlled experiment
Range sliders, 147
Raskin, Jef, 550
Raster scan, 26
RasterOp, 516. *See also* BitBlt
Rate controlled joystick, 99
Rate-controlled scrolling, 143–144, 566
RCX code, 356
Reaction Handler, 51
ReactJS, 489, 494, 520
Recognition accuracy, 366
Recognition algorithms, 629
Recommender systems, 455
Red-green color blindness, 428
Red-green-blue (RGB), 279
Redo, 383–384, 392–393
  considerations for, 411–412
  history of, 385–388
Reframe in Smalltalk, 311
Rehabilitation Engineering and Assistive Technology Society of North America (RESNA), 632
Reid, Brian, 235
Rekimoto, Jun, 289, 317
Relative coordinates, 106
Release (mouse button), 93
Remote interaction, 104–105
Rendering, 325
Repeat, 392–393. *See also* Redo
Repetitive strain injury. *See* Repetitive stress injury (RSI)
Repetitive stress injury (RSI), 248, 427, 528
Research Character Generator (RCG), 546
  RCG user study, 546
Research in Motion (RIM), 79–81, 121, 188, 196, 208
Resource Editor, 57, 487, 499
Resource files, 483, 497–498
Resource fork, 497
Resource Manager, 587
Responsible AI, 460
Responsive (web pages), 259
Retained object model, 518–519
Reverse Polish Notation (RPN), 64, 263
Reverse video, 173
RGB, 279
Ribbon, 165, 253–254
Rich text. *See* Formatted text
Right-down (mouse button), 93
RIM. *See* Research in Motion
RIM Blackberry, 79–81, 121, 188, 196, 208
Roberts, Larry, 55

Robertson, George, 473
Robots, turtles to, 355–358
Roll, 329
Roller menu. *See* Spin Menu
RollerMouse Red Plus from Contour, 113, 653
Rooms, 313, 323, 468
Rosen, Ben, 572
Ross, Doug, 54
Rotating, 119
Rotation handle, 268
Rubber band lines, 29, 271
Rubine, Dean, 146, 367, 372, 374–376
Russel, Slug, 54

Samsung, 14, 363, 456, 564, 566–567
Samsung SmartThings, 363
Sapphire, 11, 159, 161, 297–299, 304–306, 310–311
Sassafras, 512
Satisfaction, 6, 527
Scalable Vector Graphics (SVG), 518
Scanning keyboard, 210, 430–431
Schwarz, Julia, 641
  devices, 642
  HoloLens 2 interaction model, 642–643
  HoloLens 2 prototypes and interactions, 643–646
  MRTK, 646–647
  short bio of, 641–642
Screen reader, 433–435
ScreenTips, 415. *See also* Tooltip
Scribble (for Apple Watch), 215, 227
Scribble curve, 215, 276
Scribe (text formatter), 235
Script. *See* Macro
Script Model, 401–403, 410
Scroll wheel, 107, 121, 123, 143, 146, 150, 344, 427, 623, 625, 653
Scrollbar, 17, 32, 44, 57, 119, 126, 256, 297, 310, 317, 325, 349, 350, 366, 414, 486, 495, 520, 534, 549, 557, 560, 566, 578, 594, 654, 656
  Apple Lisa (1983) and Macintosh (1984), 133–135
  Bravo's Scrollbar (1974), 127–128
  Interlisp-D, Symbolics Lisp Machine, XDE Scrollbars and Andrew system, 129–131
  iPhone, 138
  mobile device, 137
  Smalltalk's Scrollbar (1977), 128–129
  variations on, 138–141
  Windows, OpenLook, and NeXT, 135–137
  Xerox Star (1981) and Viewpoint (1985), 131–133
Scrolling, 119, 578
  auto-scroll, 120–121
  considerations, 153–155
  gestures for scrolling, 144–146
  interaction design for, 566
  keys and physical buttons, 125–126
  knobs, 121–123
  rate-controlled scrolling, 143–144
  research on, 149–153
  scrollbars, 126–141
  sliders, 146–148
  speed, 149–152
  using "hand", 141–142

ScrollLock, 432
ScrollTest, 151
Sculley III, John, 68, 452
Seeheim Model, 511
Seesoft system, 139
Segmentation problem, 203
Selected state of objects, 181
Selecting objects
  brief history of graphical editing, 284–287
  considerations for, 290–291
  in consumer electronics, 282–284
  creating new objects and painting, 264
  graphical object hierarchies, 263
  of graphics, 264–269
  in linear and hierarchical lists, 282
  research in selection and editing objects, 288–290
  specifying animations and behaviors, 280–281
  specifying colors, 278–279
  specifying graphical properties, 279–280
  specifying position, size, and shape, 269–278
  in spreadsheets, 281–282
  syntax of operations, 261–263
Selection, 91–92, 392
  clicking on objects, 265–266
  feedback for selecting objects, 267–268
  of graphics, 264
  handles, 267
  mode, 264
  one or multiple objects, 264
  pixels for painting programs, 268–269
  in region, 266–267
  rules, 479
  text on desktops, 238–241
  text on mobile devices, 241–243
Selective undo, 384, 399–400, 406
  inverse model, 403–405
  by region, 400–401
  script model, 401–403
Selective-undo/-redo/-repeat method, 409
Selker, Ted, 100, 619
  origins of pointing stick, 620–622
  refining TrackPoint, 622–625
  short bio of, 619–620
Semantic, 475
Semantic differential test, 564
Semantic level model, 473–476
Semantic merge, 362
Semantic web, 320
Semantic zooming, 315
SensAble Technologies. *See* PHANTOM (6D haptic device)
Sensing technologies, 637
Sensors, 80, 102, 106, 116, 181, 229, 328, 332, 334, 337, 339–340, 350–351, 353–364, 642
Service design, 82
Shape writing, origins of, 628
ShapeWriter, 214, 369, 627, 629
  research, 369
  text entry method, 627
SHARK text entry, 214, 369, 628
Shell software, 50, 642
Shift (mouse modifier), 93
Shift-clicking, 264

Shneiderman, Ben, 52, 54, 147, 458, 460–461, 615–616
Shortcut keys. *See* Keyboard shortcuts
Shoup, Dick, 34
SHRDLU, 443
Side-scrolling games, 343
Sierpiñski Gasket (macro), 410
SIGGRAPH Video Review, 13
Silicon Graphics, Inc. (SGI), 24–25, 309, 510
SILK, 372, 506
SimCity, 615
Simonyi, Charles, 32–33, 602
Sims, the game, 615–616
Singer/Link, 55
Single Level Undo, 384, 393–394
Single window, 298–303, 309
Singularity (AI), 458
Sip-and-puff device, 429–430
Siri (iPhone), 441, 444
Situational impairments, 425
Six degrees of freedom (6DOF), 330
SketchPad, 28–30, 94, 284, 316, 493
Skeuomorphic design, 17, 287, 296, 526
Skeuomorphism, 17, 526
Slider, 119, 146–148
  research on, 149–153
Smalltalk, 11, 13, 31–32, 128, 183, 186, 295, 520, 549, 553, 576, 586–587, 654
  cursor, 239
  scrollbar (1977), 128–129
  system, 576
Smalltalk-80 version, 32
Smart Compose, 221
Smart guides, 278
Smart Home, 362
Smart selection, 241
Smart speakers, 162, 443
SmartCell technology, 223
Smartphones, 21, 63–64, 71, 232, 303, 354, 652
  interaction techniques, 651–652
SmartThings ecosystem, 363
Smartwatch interaction techniques, 77, 102, 354, 629, 652, 655
Smith, David Canfield, 37, 303, 551–552
  creation by copying, 556
  property sheets, 556
  pygmalion, 553
  star designs, 557
  story of icons, 552–553
  techniques for reducing system commands, 555
  universal commands, 555–556
  Xerox Star, 553–555
Snake game, 343
Snap, 312
Snapping, 276–278, 286, 372
SOAR, 479
Soft key, 125, 168, 576
Soft keyboard. *See* Virtual keyboard
Software architectures, 511
Software development kit (SDK), 337, 485
Software Garden Inc., 569
Software models, 466
Software patterns, 510, 537
Software tools, 466
Sony MagicLink, 70
Sony QRIO, 358
SOS Interface, 57, 499
SpaceWar, 54
Spatial Data Management System (SDMS), 11, 36, 142, 316

Spatial model, 629
Spatial reasoning, 331
Spatial visualization ability, 331
Special Interest Group on Computer-Human Interaction (SIGCHI), 7–8, 13
Special-purpose interaction technique, 652
Special-purpose systems, 444
Speech bubble, 415
Speech generation, 443
Speech interfaces, 444
Speech language
  dialog systems, 444
  feedback, verifying, and correcting, 447–448
  interfaces, 443–450
  kinds of dialogs, 444–445
  system actions, 449–450
  users provide input, 445–447
Speech recognition, 442
Speed, 227–228, 434
  of pointing, 110–115
  of text entry, 196, 232
Spell checkers, 218–220
Spin menu, 167
Spinal Cord Injury (SPI), 427
Spinning Beach Ball (Wait Cursor), 161
Spinning pinwheel, 161
Spline curve, 275
Spotlight, 310, 313
Spreadsheet, 53, 281, 521, 569–571
  selection in, 281–282
Spring–action interactions, 614
Springs and struts (for layout), 496
Squiggly underlines. *See* Wavy underlines
SRI. *See* Stanford Research Institute
Stanford Card Stunt Program, 544–545
Stanford Research Institute (SRI), 30, 53, 56, 96, 235
Star (Xerox), 11, 37, 45, 57, 131, 168, 170, 171, 174, 186, 209, 249, 262, 267, 271, 285, 294–296, 303, 310, 312, 386, 468, 553–557
Start button, 307
Start Menu, 604
Start Narrator, 435
State (of a UI), 8, 106–109, 469
State Transition Diagrams (STDs), 8, 469–471
StateCharts, 471
Status bar, 171, 414–415
Status line, 414
Steamer project, 57, 506
Stencils, 420–421
Stenographers, 201
Stenotype machine, 201
Step keys. *See* Arrow keys
Stereoscopy, 327
Sticky Keys, 432
Stretchy spacers, 496
Stroke (of a gesture), 203, 366
Stroustrup, Bjarne, 491
Structured editor, 237
Stuff I've Seen system, 317
Styled text, 233. *See also* Formatted text
Stylus, 62, 74, 76, 102, 105, 109, 137, 146, 203, 212–214, 241, 289, 338, 429, 533, 628
SubArctic toolkit, 57, 492
Submenu, 164, 172

Sugilite (multi-modal system), 448, 451, 457
Summagraphics, 101
Sun Microsystems, 24, 44, 47, 309, 566
SunOS, 25
SunWindows, 44–45, 297, 304
Super Mario Bros., 343
SuperCalc, 571
SuperPaint, 34
SUPPLE, 258
Surface Hub (Microsoft), 109
Surveillance capitalism, 460
Sutherland, Ivan, 28–29, 51, 55, 332
Sutherland, William Robert (Bert), 30
SVG (Scalable Vector Graphics), 518, 519
SwiftKey (auto-correct product), 224
Swing, 491
Switch. *See* Toggle Switch
Switchbacks, 151
Sword of Damocles, 332–333
Swype, 214
Symbian, 85, 629
Symbolics, 24, 130–131, 187, 252, 295
Symbolics Lisp Machine, 129–131
Symbolics window manager, 187
Syntactic level model, 473–476
Syntax, 237, 261, 447, 475
  of operations, 261
Syntax-directed editor. *See* Structured Editor
System Usability Scale (SUS), 528

T9, 71, 207–208, 467, 652
Tab key (text input fields), 50, 246, 260, 513
Tabbed dialog, 249, 252–253
Tabbed document interface (TDI), 303
Tabbed windows, 66, 298–303
tableHeaderView, 610
Tablet, 21, 60–62, 85, 100–102, 106, 369, 561, 654
Tabletop User Interfaces, 102, 109, 354, 358–359
TabletPC, 81, 85, 107, 369
Tabs, 66
Tactile feedback. *See* haptic feedback
Tajo. *See* Xerox Development Environment
Tangible Media Group, 359
Tangible user interfaces (TUIs), 354
Tap (event), 93–94
Tap (text entry device), 216
Tap Systems, Inc., 216
TapSense, 639
Target text for testing, 231–232
Taskbar, 161, 295, 304–309, 312
Tear-Off Menu, 166–167, 183
TECO, 53
Teddy, 347, 372
Teitelman, Warren, 385
Teletype machine, 23
Tendonitis. *See* Repetitive stress injury (RSI)
Terminals (display), 26, 50
  window, 28
Tesler, Larry, 2, 32, 38, 244, 250, 387, 544, 546, 586
  blank screen study, 545–546
  cut and paste, 546–547
  gypsy text editor, 548
  interaction techniques at Apple, 549–550

modeless software and gypsy text editor, 547–548
problems with modes, 548–549
RCG user study, 546
short bio of, 544
user studies, 544–545
TeX, 235, 496
Texas Instruments (TI), 64, 263
Text box. *See* Text entry fields
Text buffer demo, 547
Text cursor, 91–92, 238
Text editing, 53, 194, 232, 578
brief history of, 233–236
and formatting, 232
operations, 244
techniques, 238–245
Text entry, 193, 431–432, 655
auto-fill, auto-correction, and autocomplete, 217–227
chorded text entry, 201
considerations for designing new text entry methods, 247–248
for desktops, 194–201
fields, 245
measuring text entry speed and accuracy, 227–232
method, 231, 652
for mobile devices, 201–217
text editing and formatting, 232–245
typeahead, 246–247
on wearables and watches, 215–217
widgets, 194, 245–246
Text formatting, 194, 232
languages, 233–234
Text selection, 238
Text-edit interactor, 515
TextPlus (autocomplete), 223
Textual natural language interfaces, 447
Theta Menus, 616
Think aloud (study), 528
ThinkPad laptops, 623
Three dimensions (3D), 55, 325
1D and 2D input devices for, 332
3D-IxTs difficult, 328–331
3D-touch, 93
considerations for 3D interactions, 349–351
image, 327
interaction techniques, 340–341, 652
interactions, 510
IxTs for games, 341–344, 655
IxTs using mouse, 344–349
models, 467
objects, 261
pointing devices, 91
pointing test, 652
position sensors and HMDs, 332–333
press, 242
print physical widgets, 359
rendering, 294
tools for 3D interactions, 510
touch, 93, 109
widgets, 349
Three Rivers Computer Corporation, 11, 24–25, 159
Three-dimensional interaction technique (3D-IxT), 325
input devices for, 332–340
Thumb (finger), 80, 93, 103, 208, 217, 428, 620, 652
Thumb (of scrollbar), 126–137, 520, 578

Tiled windows, 294–295, 299–300
Tiling systems, 302
Tilt-to-scroll, 152, 340
Time-machine computing, 317
Timed response test, 564
Timeline of major systems, 87
Timeline systems, 317–318
Timeline View, 317
TimeScape system, 317
Timesharing, 22, 27, 50
Title line, 250, 297, 301, 309–313
Toggle Switch, 168, 175, 178
Tokens, 237, 240, 475
Tool metaphor, 264, 468
Toolbar, 253–254, 415
Toolbox, 490
Toolglass, 186
Toolkits, 56–57, 282, 486
  creators, 491
Tooltip, 171, 414–417, 419
Topaz, 288, 410
TopView, 600
Touch, 93
  groups, 377
  increasing richness of touch interfaces, 639
  sensors, 639
  systems, 579
Touch (event), 93
Touch Bar, 102
Touch type, 195
Touch-and-drag, 93
Touch-and-hold, 93
Touch-down, 93
Touch-up, 93
Touchpad, 61, 146
Touchscreen, 61, 102–104, 641
Trace Center, 424, 432, 438–439, 632
Track, 126
Track bar. *See* Slider
Trackball, 97, 99–100
TrackPoint, 100, 143, 149, 619
  refining, 622–625
Trade secret, 531, 534
Trademark, 531, 533
Transition (between states), 8
Transition Diagrams. *See* State Transition Diagrams
Transitions, 469
Tree model, 397–398
Tree view. *See* Hierarchical view
Treo (Palm), 76
Trillium, 57, 506
Triple clicks, 93
Troff, 235
Troll (Patent). *See* Non-Practicing Entity (NPE)
Trough (of scrollbar), 126, 495
Trower, Tandy, 597
  graphic design, 601–602
  Microsoft Bob and Windows assistant, 604–605
  Microsoft products, 605–607
  Microsoft Windows 1.0, 2.0, and 3.0, 598–601
  Microsoft's user interface design group and visual basic, 602–603
  precursors to Microsoft Windows, 598
  short bio of, 597
  Windows NT and Windows 95, 603–604
Turing Test, 443
Turing, Alan, 443
Turtle (LOGO), 355–358
TVEdit, 53
Tweetie, 609–610

beta version of, 612
Tweetie 2, 614
Twiddler, 208, 215
Twitter apps, 609, 614
Two dimensions (2D), 325
input devices for 3D, 332
layout, 495
objects, 261
pointing devices, 91
rendering, 294
scrolling, 119
Two-finger scroll, 145–146
Two-finger swipe, 639
Two-handed interfaces, 655
"Two-state" design, 613
TX-2, 28, 121, 371, 560, 653
Typeahead, 246–247
Typesetting. *See* Text formatting
Typewriters, 194–195

Ubicomp. *See* Ubiquitous Computing
Ubiquitous Computing (UbiComp), 31, 64–65, 353
UI. *See* user interface
UIMS. *See* User Interface Management System
Undergraduate Research Opportunities Program (UROP), 11
Undo, 16, 219, 244, 383–384, 391–393
back button, 405–406
considerations for, 411–412
history of, 385–388
implementing undo and command object pattern, 408–410
macro recording and programming by example, 410–411
multi-level undo, 394–398
multi-user undo, 406–408
operation, 383
scope of, 408
selective undo, 399–405
single-level undo, 393–394
unit of, 390–391
Undo, cut, copy, paste (ZXCV), 583
Undo model, 384, 466
Undo stack, 384
Undoes-the-undo, 392
Unified Modeling Language (UML), 471
Uniform Resource Locator (URL), 47
Unipad (autocomplete), 223
Unistrokes, 65, 76, 203, 366, 369, 378
United States Patent and Trademark Office (USPTO), 533, 534
Unity, 510, 616, 643, 652
Universal Clipboard, 289
Universal commands, 555–556
Universal design, 425
Unreal Engine, 510, 643
Up (mouse button), 93
Urban Planning system (URP system), 359
US Air Force, 28
US Department of Defense (DOD), 51
Usability, 6–7
testing, 527
Usability evaluation, 527–528
User, 5–6
early tracking of user's fingers, 333–335
model, 466
studies, 544–545
testing, 581

User Experience (UX), 6, 488
User interface (UI), 6–7, 367, 483, 657
  for conversational agents, 654–655
  Design Group, 603
  design patterns for UIs, 467–469
  framework, 486
  interactive and graphical tools for building, 499–503
  metaphors and UI design patterns for, 467–469
  software tools, 485
  toolkit, 486
User Interface Management System, 56–57
User Interface Software and Technology conference (UIST), 8, 379, 628
User interface technique. *See* Interaction technique
User model, 466
User Systems Ergonomics Research, 619
UX. *See* User Experience

van Dam, Andy, 53, 385, 475
Vanderheiden, Gregg, 424, 631
  accessibility, 635–636
  digital affinity, 632–634
  info-bot, 636
  morphic, 634–635
  personal history, 632
  short bio of, 631–632
Vaporware, 599
Vector scan display, 26
Vernier effect, 561
Very high frequency band (VHF band), 185
vi (Text editor), 234, 236, 238
Vicon (tracker), 336
Video games, 54, 336, 341–344, 364
Videocassette recorder (VCR), 187, 209
VideoPlace, 146
Viewpoint (Xerox), 131–133, 295, 302
Viewport, 294
Vimeo, 148
Virtual assistant. *See* Conversational agent
Virtual desktop, 313, 323
Virtual devices, 471–472
Virtual keyboard, 209–212, 628
  gestures over virtual keyboards, 213–215
  research on virtual keyboard layouts, 212–213
Virtual knob, 121
Virtual reality (VR), 55–56, 327, 332, 334, 347, 350–352, 372, 510, 635. *See also* Augmented reality (AR)
  IxTs for, 655
  systems, 372
Virtual reality/augmented reality (VR/AR), 424
VisiCalc, 35–36, 53, 295, 569–572
Vision disability. *See* Visual impairment
Visual Basic (VB), 220, 360, 487, 500
  Microsoft's user interface design group and, 602–603
Visual icon design, 564–565
Visual impairment, 427–428
Visual programming, 29–30, 356, 358
Visual semantics, 552

Visual Studio (Microsoft), 10, 140, 220–221, 236, 303, 487
Visual-spatial ability, 331
Visualization, 11, 264, 317, 398, 458
Voice assistant, 162, 363, 443
von Ahn, Luis, 438

Wacom, 101, 654
Wallace, Victor, 471–472
WAP. *See* Wireless Access Protocol
Warnock, John, 34–35, 263
Watch, 76, 152, 215, 304, 305, 361
Watchboard (text entry for a watch), 216, 652
Watches, text entry on, 215–217
WatchWriter (text entry for a watch), 216
WaveLAN, 73
Wavy underlines, 219
Wearable devices, 62, 121, 215, 355, 361
Wearables, 61–62, 361
  computing, 355
  technology, 355
  text entry on, 215–217
Web. *See* World Wide Web
Web accessibility, 437–439
Web Accessibility Initiative (WAI), 438
Web browsers, 47, 485–486
Web Content Accessibility Guidelines (WCAG), 438, 528
Web pages as forms, 255
Weiser, Mark, 64, 541, 615–616
What you See is what you get. *See* WYSIWYG
Widget, 9, 51
WiFi, 73
Wiggle (gesture), 244
Wii, Nintendo, 335
Wikipedia, 9, 653
Window manager (WM), 52, 293, 296, 300
  alternatives, 315–321
  brief history of, 294–296
  collections of windows, 313–315
  considerations for, 321–323
  decorations for individual windows, 309–310
  icons, 303–306
  input focus, 296–298
  interaction techniques for windows, 310–313
  overlapping windows *vs*. tiled *vs*. tabbed *vs*. single window, 298–303
  task bar and dock, 306–309
Window system, 294
Windows, 109, 252, 293, 578, 597
  collections of, 313–315
  computers, 59
  full screen, 311
  interaction techniques for, 310–313
  scrollbars, 135–137
  system, 294, 310, 634
Windows (Microsoft), 40, 52, 93, 96, 110, 135, 143, 161, 171, 178, 183, 187, 252, 261, 295, 297, 298, 310, 387, 435, 443, 485, 598, 644
Windows 1.0, 414
Windows 2.0, 600, 602
Windows 3.0, 600
Windows 6 Vista, 606
Windows 10, 297
Windows 95, 603–604

Windows Assistant, 604–605
Windows CE, 78–80, 188, 369
Windows For Pen, 68
Windows Media Center, 605
Windows Mobile, 629
Windows NT, 306, 603–604
Windows TabletPC, 81
Windows XP, 605
Winograd, Terry, 443
Wireframes, 504
Wireframing tools. *See* Prototyping Tools
Wireless Access Protocol (WAP), 48, 71
Wireless Andrew, 73
Wireless networks, 73–74
Wirth, Niklaus, 491
Wobbrock, Jacob, 378, 380
Word. *See* Microsoft Word
Word completion. *See* Autocomplete
Word prediction. *See* Autocomplete
Word processor, 235
Words per minute (WPM), 193, 196, 207, 227
WordVision, 303
Working set, 237, 313
World view, 316
World Wide Web, 46–48, 54, 71, 303, 320, 438, 632
  Accessibility guidelines, 528
World Wide Web Consortium (W3C), 438
WPM. *See* Words per minute
WYSIWYG, 33, 38, 53, 235, 385

X window manager, 45, 52, 294, 296, 313, 429, 490
X/11 system, 512
X10 windowing system, 616
XAML, 498
xBox, 336
XDE. *See* Xerox Development Environment
Xerox Alternate Reality Kit system, 187
Xerox Alto, 31
Xerox Corporation, 37
Xerox Development Environment (XDE), 35, 130, 186
  scrollbars, 129–131
Xerox Palo Alto Research Center (PARC), 11, 24, 31, 42, 51, 56, 64, 96, 127, 186, 201, 235, 286, 288, 295, 305, 313, 385, 387, 544, 545, 553, 560, 563, 579, 585–587, 598, 602
Xerox programming environments, 35
Xerox Star, 24, 37–38, 57, 131–133, 171, 186, 295, 532, 553–555, 565
  computer, 566
  property sheets, 249–250
Xerox Star (1981), 598
  interface, 598
Xt, 490
XWand, 335

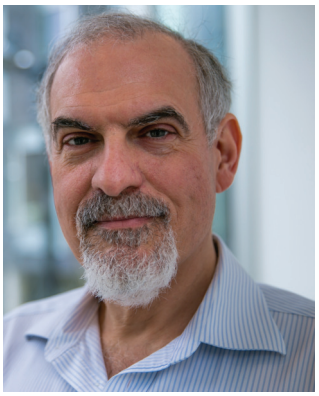
Yahoo, 223
Yaw, 329
YouTube, 148

Zhai, Shumin, 627
  algorithms, 629–630
  effective text entry requirements, 628
  origins of shape writing, 628
  pixel squeeze-to-assist in Google Pixel 2 Phone, 630

ShapeWriter, 629
short bio of, 627–628
Zhuyin method, 198
Zimmerman, Tom, 334
ZOG, 54
Zoomable User Interface (ZUI), 11, 29, 36, 315–316
PAD, PAD++, and Jazz, 316–317
SDMS, 316
SketchPad, 316
ZoomBoard (text entry for a watch), 216
Zooming, 119, 315, 591, 595
Zuckerberg, Mark, 459

Author's Biography

Brad A. Myers



Brad A. Myers has been designing, implementing, and documenting interaction techniques for over 40 years. He is the Charles M. Geschke (SCS 1973) Director of the Human–Computer Interaction Institute and professor in the School of Computer Science at Carnegie Mellon University, with an affiliated faculty appointment in the Software and Societal Systems Department.

He was chosen to receive the 2017 ACM SIGCHI Lifetime Achievement Award in Research for outstanding fundamental and influential research contributions to the study of human–computer interaction, and he was awarded the 2022 Alan J. Perlis Award for Imagination in Computer Science "for pioneering human-centered methods to democratize programming" from the School of Computer Science, Carnegie Mellon University. He is an IEEE Fellow, ACM Fellow, member of the CHI Academy, and winner of 18 best paper type awards and six most influential paper awards. He is the author or editor of over 550 publications, including the books *Creating User Interfaces by Demonstration* and *Languages for Developing User Interfaces*, and he has been on the editorial board of six journals. He has been a consultant on user interface design and implementation to over 90 companies, and he regularly teaches courses on user interface design and software. Myers received a Ph.D. in computer science at the University of Toronto where he developed the Peridot user interface tool. He received the M.S. and B.Sc. degrees from the Massachusetts Institute of Technology during which time he was a research intern at Xerox PARC. From 1980 until 1983, he worked at PERQ Systems Corporation. His research interests include user interfaces, programming environments, programming language design, end-user software engineering (EUSE), API usability, developer experience (DevX or DX), interaction techniques, programming by example, mobile computing, and visual programming. He belongs to ACM, SIGCHI, IEEE, and the IEEE Computer Society.

Pick, Click, Flick!

The Story of Interaction Techniques

Brad A. Myers

This book provides a comprehensive study of the many ways to interact with computers and computerized devices. An “interaction technique” starts when the user performs an action that causes an electronic device to respond, and includes the direct feedback from the device to the user. Examples include physical buttons and switches, on-screen menus and scrollbars operated by a mouse, touchscreen widgets, gestures such as flick-to-scroll, text entry on computers and touchscreens, input for virtual reality systems, interactions with conversational agents such as Apple Siri, Google Assistant, Amazon Alexa, and Microsoft Cortana, and adaptations of all of these for people with disabilities. *Pick, Click, Flick!* is written for anyone interested in interaction techniques, including computer scientists and designers working on human-computer interaction, as well as implementers and consumers who want to understand and get the most out of their digital devices.

REVIEWS

“*Pick, Click, Flick!* is an impressive reference manual of the many years of interaction design development. It is a reference book, invaluable when questions arise, whether while you are busy designing something, or learning, or teaching, where assigning sections of the reference will be a valuable resource and learning tool for students. Brad Myers has provided a great service to the interaction community.” - *Don Norman, Distinguished Prof. Emeritus, Design Lab, University of California, San Diego*

“Every UX professional should immerse themselves in this book. Not only does it unravel the fascinating and complex history of GUI widgets that will captivate any user interface nerd, but it also stands as the definitive guide to an incredibly diverse array of interaction techniques. This is not just an engaging read; it’s an essential toolkit.” - *Jakob Nielsen, Principal, Nielsen Design Group*

ABOUT THE AUTHOR

Brad A. Myers is the Charles M. Geschke Director and Professor in the School of Computer Science at Carnegie Mellon University. In 2017, he received the ACM SIGCHI Lifetime Achievement Award in Research for outstanding fundamental and influential research contributions to the study of human-computer interaction.

ABOUT ACM BOOKS

ACM Books is a series of high-quality books published by ACM for the computer science community. ACM Books publications are widely distributed in print and digital formats by major booksellers and are available to libraries and library consortia. Individual ACM members may access ACM Books publications via separate annual subscription.

BOOKS.ACM.ORG

ISBN 979-8-4007-0947-0