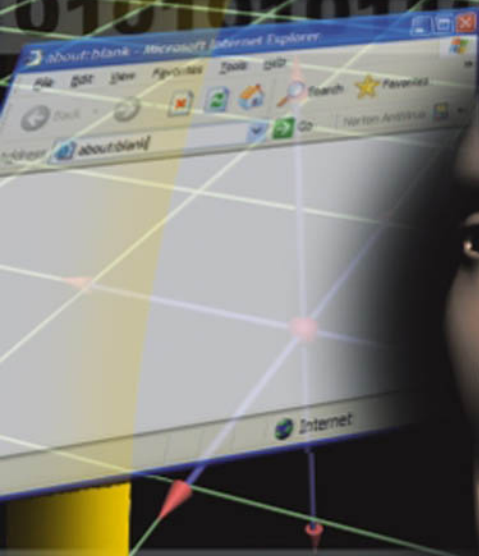


WORDWARE GAME DEVELOPER'S LIBRARY



# DirectX 9<sup>®</sup> User Interfaces

*Design and  
Implementation*



CD-ROM  
Included



**Alan Thorn**

# **DirectX<sup>®</sup> 9 User Interfaces: Design and Implementation**

**Alan Thorn**

This page intentionally left blank.

# **DirectX<sup>®</sup> 9 User Interfaces: Design and Implementation**

**Alan Thorn**

**Wordware Publishing, Inc.**

**Library of Congress Cataloging-in-Publication Data**

Thorn, Alan.

DirectX 9 user interfaces : design and implementation / by Alan Thorn.

p. cm.

ISBN 1-55622-249-1 (pbk.)

1. User interfaces (Computer systems). 2. DirectX. I. Title.

QA76.9.U83T53 2004

005.4'38—dc22

2003025230

CIP

© 2004, Wordware Publishing, Inc.

All Rights Reserved

2320 Los Rios Boulevard  
Plano, Texas 75074

No part of this book may be reproduced in any form or by any means  
without permission in writing from Wordware Publishing, Inc.

Printed in the United States of America

ISBN 1-55622-249-1

10 9 8 7 6 5 4 3 2 1  
0401

DirectX is a registered trademark of Microsoft Corporation in the United States and/or other countries. Microsoft DirectX 9.0 SDK Update (Summer 2003) copyright Microsoft Corporation, 2002. All rights reserved. All brand names and product names mentioned in this book are trademarks or service marks of their respective companies. Any omission or misuse (of any kind) of service marks or trademarks should not be regarded as intent to infringe on the property of others. The publisher recognizes and respects all marks used by companies, manufacturers, and developers as a means to distinguish their products.

This book is sold as is, without warranty of any kind, either express or implied, respecting the contents of this book and any disks or programs that may accompany it, including but not limited to implied warranties for the book's quality, performance, merchantability, or fitness for any particular purpose. Neither Wordware Publishing, Inc. nor its dealers or distributors shall be liable to the purchaser or any other person or entity with respect to any liability, loss, or damage caused or alleged to have been caused directly or indirectly by this book.

All inquiries for volume purchases of this book should be addressed to Wordware Publishing, Inc., at the above address. Telephone inquiries may be made by calling:

(972) 423-0090

# Dedication

To my mother, Christine; my father, Gary; and my sister,  
Angela.

This page intentionally left blank.

# Contents

	Acknowledgements . . . . .	xv
	Introduction. . . . .	xvii
Chapter 1	<b>User Interfaces . . . . .</b>	<b>1</b>
	1.1 User Interfaces — What Are They? . . . . .	2
	1.2 Controls — Gadgets and Gizmos . . . . .	4
	1.2.1 Text Boxes. . . . .	5
	1.2.2 Text Edits . . . . .	6
	1.2.3 Buttons . . . . .	7
	1.2.4 Labels . . . . .	8
	1.2.5 List Boxes . . . . .	9
	1.2.6 Drop-Down Lists. . . . .	10
	1.2.7 Check Boxes . . . . .	11
	1.2.8 Menus . . . . .	12
	1.2.9 Page Controls/Tab Controls . . . . .	13
	1.2.10 Windows and Other Containers . . . . .	14
	1.3 Interface Flow Diagrams — Interfaces on Paper . . . . .	15
	1.4 Interface Design — Tips and Tricks. . . . .	16
	1.4.1 Be Consistent . . . . .	17
	1.4.2 Know Your Audience. . . . .	17
	1.4.3 Justification and Alignment . . . . .	18
	1.4.4 Grouping Data . . . . .	19
	1.4.5 Error Handling. . . . .	19
	1.4.6 Disabling Program Features . . . . .	20
	1.4.7 Graphics, Colors, Icons, and Art. . . . .	21
	1.4.8 Balancing Text and Symbols . . . . .	21
	1.4.9 Paths and Navigation. . . . .	22
	1.4.10 Keyboard Support. . . . .	24
	1.4.11 Tool Tips . . . . .	24
	1.5 Conclusion . . . . .	24
Chapter 2	<b>Introducing DirectX . . . . .</b>	<b>27</b>
	2.1 DirectX — What Is It? . . . . .	28
	2.1.1 Direct3D — Graphics . . . . .	29
	2.1.2 DirectInput — Keyboards, Mice, and Joysticks . . . . .	29
	2.1.3 DirectMusic and DirectSound — MIDI and WAV. . . . .	30

	2.1.4 DirectPlay — Networking . . . . .	30
	2.1.5 DirectShow — Programmable Media Player . . . . .	30
	2.2 DirectX — Other Features . . . . .	31
	2.2.1 Mesh Viewer . . . . .	31
	2.2.2 Error Lookup. . . . .	32
	2.2.3 Caps Viewer . . . . .	33
	2.2.4 GraphEdit . . . . .	34
	2.2.5 Texture Tool . . . . .	35
	2.3 System Requirements . . . . .	36
	2.4 Where to Obtain DirectX . . . . .	36
	2.5 Installation . . . . .	37
	2.6 Installed Files . . . . .	38
	2.7 Configuring Visual C++ . . . . .	39
	2.8 Coding with Hungarian Notation . . . . .	42
	2.9 Conclusion . . . . .	43
Chapter 3	<b>Introducing Direct3D . . . . .</b>	<b>45</b>
	3.1 Direct3D Concepts — Overview and Mathematics. . . . .	46
	3.2 Getting Started. . . . .	48
	3.3 Programming Direct3D Applications . . . . .	51
	3.4 Initializing Direct3D . . . . .	52
	3.5 Creating a Direct3D Device — A Graphics Card . . . . .	53
	3.6 Preparing to Render . . . . .	56
	3.7 Initializing World Data . . . . .	58
	3.7.1 Direct3D Surfaces — IDirect3DSurface9 . . . . .	59
	3.7.2 Direct3D Surfaces — Loading Image Files . . . . .	60
	3.7.3 Direct3D Surfaces — Rendering . . . . .	62
	3.7.4 Direct3D Textures — IDirect3DTexture9. . . . .	65
	3.7.5 Direct3D Textures — Preparing to Render . . . . .	66
	3.7.6 Direct3D Textures — Rendering . . . . .	67
	3.8 Alpha Blending. . . . .	72
	3.8.1 Using Adobe Photoshop . . . . .	73
	3.8.2 Using Paint Shop Pro . . . . .	74
	3.8.3 Using the DirectX Texture Tool . . . . .	75
	3.8.4 Enabling Alpha Blending in Direct3D . . . . .	76
	3.9 Conclusion . . . . .	77
Chapter 4	<b>Introducing DirectInput . . . . .</b>	<b>79</b>
	4.1 DirectInput Basics . . . . .	80
	4.2 Getting Started. . . . .	80
	4.3 Programming. . . . .	82
	4.4 Creating a DirectInput Object . . . . .	83
	4.5 Creating DirectInput Devices . . . . .	85
	4.6 The Keyboard . . . . .	86
	4.6.1 Creating the Keyboard. . . . .	86
	4.6.2 Configuring the Keyboard . . . . .	87

	4.6.3 Reading from the Keyboard . . . . .	90
	4.7 The Mouse . . . . .	92
	4.7.1 Creating the Mouse . . . . .	92
	4.7.2 Setting the Cursor . . . . .	93
	4.7.3 Reading from the Mouse. . . . .	95
	4.7.4 Processing the Cursor Position . . . . .	96
	4.7.5 Reading Mouse Buttons . . . . .	97
	4.8 Conclusion . . . . .	98
<b>Chapter 5</b>	<b>Wrapping Direct3D . . . . .</b>	<b>99</b>
	5.1 CXSURFACE — Wrapping Surfaces . . . . .	100
	5.1.1 Instantiating and Deleting CXSURFACE . . . . .	101
	5.1.2 Loading Images . . . . .	102
	5.1.3 Copying Surfaces . . . . .	102
	5.1.4 Representing the Back Buffer . . . . .	103
	5.1.5 Rendering. . . . .	104
	5.1.6 Using CXSURFACE . . . . .	104
	5.2 CXTEXTURE — Wrapping Textures . . . . .	106
	5.2.1 Instantiating and Deleting . . . . .	107
	5.2.2 Loading Images. . . . .	107
	5.2.3 Preparing to Render . . . . .	108
	5.3 CXPEN — Wrapping ID3DXSPRITE . . . . .	109
	5.3.1 Instantiating and Deleting . . . . .	109
	5.3.2 Rendering Textures. . . . .	110
	5.3.3 Using CXPEN and CXTEXTURE. . . . .	111
	5.4 Conclusion . . . . .	112
<b>Chapter 6</b>	<b>Abstracting DirectInput . . . . .</b>	<b>113</b>
	6.1 CXINPUT — The DirectInput Object . . . . .	114
	6.1.1 Instantiating the DirectInput Object . . . . .	115
	6.1.2 Creating Input Devices . . . . .	116
	6.2 CXKEYBOARD — Wrapping the Keyboard Device. . . . .	117
	6.2.1 Instantiating Keyboard Devices . . . . .	118
	6.2.2 Reading from CXKEYBOARD . . . . .	119
	6.3 Wrapping the Mouse Device . . . . .	121
	6.3.1 CXMOUSESURFACE — Wrapping a List of Cursors . . . . .	121
	6.3.2 Linked Lists — A Definition . . . . .	122
	6.3.3 Navigating Linked Lists . . . . .	123
	6.3.4 Adding New Items to Linked Lists. . . . .	124
	6.3.5 Deleting Linked Lists. . . . .	125
	6.3.6 CXMOUSESURFACE — Other Properties. . . . .	125
	6.3.7 Wrapping the Mouse Device with CXMOUSE . . . . .	126
	6.3.8 Initializing Mouse Cursors with CXMOUSE . . . . .	129
	6.3.9 Changing Mouse Cursors with CXMOUSE . . . . .	131
	6.3.10 Reading Mouse Data with CXMOUSE . . . . .	132
	6.3.11 Reading Cursor Positions with CXMOUSE . . . . .	133

- 6.3.12 Reading Button States with CXMouse . . . . . 133
- 6.4 Conclusion . . . . . 134
  
- Chapter 7 **Beginning CXControl** . . . . . 135
  - 7.1 UI LIB (User Interface Library) — What Is It? . . . . . 136
  - 7.2 UI LIB — Controls as Classes . . . . . 136
  - 7.3 Controls — Class Hierarchy and Base Controls . . . . . 137
  - 7.4 CXControl — The Beginnings . . . . . 138
  - 7.5 Defining CXControl — Controls and a Canvas. . . . . 139
  - 7.6 CXControl — Parent, Sibling, and Child Controls . . . . . 140
    - 7.6.1 Adding Child Controls . . . . . 143
    - 7.6.2 Clearing Child Controls. . . . . 144
    - 7.6.3 Removing Specific Children . . . . . 145
    - 7.6.4 Counting Child Controls . . . . . 146
  - 7.7 Absolute and Relative Positioning . . . . . 146
    - 7.7.1 Computing Positions . . . . . 149
  - 7.8 CXControl — The Class Declaration Thus Far . . . . . 150
  - 7.9 Conclusion . . . . . 151
  
- Chapter 8 **Continuing CXControl** . . . . . 153
  - 8.1 Messages . . . . . 154
    - 8.1.1 Posting Messages. . . . . 157
    - 8.1.2 Message Specifics . . . . . 157
  - 8.2 Handling Mouse Messages. . . . . 158
    - 8.2.1 Cursor Intersection. . . . . 160
    - 8.2.2 Hierarchical Posting . . . . . 161
    - 8.2.3 Triggering Mouse Events . . . . . 163
  - 8.3 Handling Keyboard Messages . . . . . 164
    - 8.3.1 Focus . . . . . 165
    - 8.3.2 Triggering Events . . . . . 166
  - 8.4 Handling Control Painting . . . . . 167
  - 8.5 Posting in Reverse . . . . . 168
  - 8.6 Depth Sorting. . . . . 170
  - 8.7 Triggering Paint Events . . . . . 173
  - 8.8 CXControl — The Final Declaration . . . . . 174
  - 8.9 Conclusion . . . . . 176
  
- Chapter 9 **Developing Windows**. . . . . 179
  - 9.1 CXWindow — Deriving from CXControl . . . . . 180
  - 9.2 Desktop and Application Windows . . . . . 181
  - 9.3 Class CXWindow as a Parent. . . . . 181
  - 9.4 Implementing the Parent Window . . . . . 183
  - 9.5 CXWindow as a Child Window . . . . . 183
  - 9.6 Implementing Child Windows . . . . . 184
    - 9.6.1 Child Windows — Loading the Canvas. . . . . 185

	9.6.2	Painting Application Windows . . . . .	186
	9.6.3	Dragging Application Windows. . . . .	187
	9.6.4	Minimizing and Restoring Application Windows. . . . .	190
	9.7	Using CxWindow — Sample Application . . . . .	193
	9.7.1	Overview . . . . .	198
	9.7.2	Desktop Initialization. . . . .	198
	9.7.3	Window Initialization . . . . .	199
	9.7.4	Windows Message Posting. . . . .	199
	9.7.5	Deleting an Interface . . . . .	200
	9.8	Conclusion . . . . .	201
<b>Chapter 10</b>		<b>Labels and Buttons . . . . .</b>	<b>203</b>
	10.1	Labels and Buttons . . . . .	204
	10.2	CXLabel — Labels . . . . .	204
	10.3	Labels as ID3DXFont . . . . .	205
	10.3.1	Instantiating ID3DXFont . . . . .	206
	10.3.2	Setting the Label Caption . . . . .	209
	10.3.3	Painting with ID3DXFont . . . . .	209
	10.3.4	Releasing ID3DXFont . . . . .	212
	10.4	CXButton — Buttons . . . . .	212
	10.5	CXButton — The Class Declaration . . . . .	213
	10.5.1	The Class Constructor . . . . .	214
	10.5.2	Setting Pressed and Unpressed Images . . . . .	215
	10.5.3	Setting the Button Caption . . . . .	216
	10.5.4	Painting . . . . .	217
	10.5.5	Destructor. . . . .	218
	10.6	CXLabel and CXButton — A Sample Application . . . . .	219
	10.7	Conclusion. . . . .	224
<b>Chapter 11</b>		<b>Text Boxes and Check Boxes . . . . .</b>	<b>225</b>
	11.1	Text Boxes and Check Boxes . . . . .	226
	11.2	Text Boxes . . . . .	226
	11.3	Clever Strings — Std::String . . . . .	227
	11.3.1	Initialization and Assigning . . . . .	228
	11.3.2	String Lengths. . . . .	229
	11.3.3	Editing and Appending Strings . . . . .	229
	11.3.4	Copying Substrings . . . . .	230
	11.3.5	Converting Strings to char*. . . . .	230
	11.3.6	Erasing and Emptying. . . . .	231
	11.4	Lines — ID3DXLINE . . . . .	231
	11.4.1	Drawing Lines. . . . .	232
	11.5	CXTextBox — The Class Declaration . . . . .	233
	11.5.1	Constructor . . . . .	234
	11.5.2	Text Width and Height . . . . .	235
	11.5.3	Setting Text . . . . .	236
	11.5.4	Text Box Caret . . . . .	236

11.5.5	Inserting Text . . . . .	237
11.5.6	Removing Text . . . . .	238
11.5.7	Processing Keypresses . . . . .	238
11.5.8	Cursor Positioning. . . . .	240
11.5.9	Caret at Cursor . . . . .	241
11.5.10	Handling the Mouse . . . . .	242
11.5.11	Painting. . . . .	242
11.5.12	Cleaning Up . . . . .	243
11.6	Check Boxes . . . . .	244
11.7	CXCheckBox — The Class Declaration . . . . .	244
11.7.1	Image and Text Loading. . . . .	245
11.7.2	Checking and Unchecking. . . . .	246
11.7.3	Painting . . . . .	246
11.7.4	Cleaning Up . . . . .	247
11.8	Conclusion. . . . .	247

**Chapter 12 Scrolling Lists . . . . . 249**

12.1	Scroll Bars, List Boxes and Drop-Down Lists . . . . .	250
12.2	CXScrollBar — Scroll Bars as a Class . . . . .	250
12.2.1	The Class Constructor . . . . .	253
12.2.2	Arrows, a Thumb, and a Background . . . . .	253
12.2.3	Width and Height, Min and Max . . . . .	255
12.2.4	Screen Positions to Scroll Values . . . . .	255
12.2.5	Scaling the Thumb . . . . .	257
12.2.6	Setting the Thumb Position. . . . .	258
12.2.7	Handling Input. . . . .	259
12.2.8	Tiling the Background. . . . .	260
12.2.9	Painting . . . . .	261
12.2.10	CXScrollBar — Cleaning Up. . . . .	262
12.3	List Boxes and List Items. . . . .	263
12.4	CXListItem — ListItem as a Class . . . . .	263
12.4.1	The Class Constructor . . . . .	265
12.4.2	Setting Item Size . . . . .	265
12.4.3	Painting . . . . .	266
12.5	CXListBox — List Boxes as Classes . . . . .	268
12.5.1	The Class Constructor . . . . .	271
12.5.2	Loading Item Backgrounds . . . . .	271
12.5.3	Loading the Scroll Bar. . . . .	272
12.5.4	Computing a List Frame . . . . .	273
12.5.5	Adding List Items . . . . .	274
12.5.6	Clearing List Items . . . . .	276
12.5.7	Getting Items by Index . . . . .	277
12.5.8	Getting Items by (X,Y) Position. . . . .	277
12.5.9	Scrolling the Frame . . . . .	278
12.5.10	Handling Input . . . . .	280
12.5.11	Painting. . . . .	281

	12.5.12 Cleaning Up . . . . .	282
	12.6 CXDropDownList — Drop-Down Lists as Classes. . . . .	283
	12.6.1 The Class Constructor . . . . .	285
	12.6.2 Initializing the Drop-Down List . . . . .	286
	12.6.3 Showing and Hiding the List . . . . .	287
	12.6.4 Handling Input. . . . .	288
	12.6.5 Painting . . . . .	289
	12.6.6 Cleaning Up . . . . .	290
	12.7 Conclusion. . . . .	290
<b>Chapter 13</b>	<b>Introducing DirectShow . . . . .</b>	<b>291</b>
	13.1 DirectShow — What Is It? . . . . .	292
	13.2 Getting Started . . . . .	294
	13.3 The Filter Graph . . . . .	295
	13.4 The Media Control . . . . .	297
	13.5 The Event Mechanism . . . . .	297
	13.6 Registering for Events . . . . .	298
	13.7 Loading a File . . . . .	299
	13.8 Playing a File . . . . .	301
	13.9 Catching Media Events . . . . .	301
	13.10 Reading Media Events . . . . .	302
	13.11 Handling Media Events . . . . .	305
	13.12 Cleaning Up . . . . .	306
	13.13 Conclusion . . . . .	306
<b>Chapter 14</b>	<b>Wrapping DirectShow . . . . .</b>	<b>307</b>
	14.1 CXMedia and CXMediaPlayer . . . . .	307
	14.2 CXMedia — Songs, Movies, and More. . . . .	308
	14.3 CXMediaPlayer — Player of the Playlist. . . . .	309
	14.3.1 The Class Constructor . . . . .	310
	14.3.2 Initializing DirectShow . . . . .	311
	14.3.3 Adding Media Files . . . . .	311
	14.3.4 Clearing Media Files . . . . .	312
	14.3.5 Playing a Playlist . . . . .	313
	14.3.6 Pausing and Stopping . . . . .	314
	14.3.7 Handling Messages . . . . .	315
	14.3.8 Uninitializing DirectShow. . . . .	316
	14.3.9 Cleaning Up . . . . .	317
	14.4 Conclusion. . . . .	317
<b>Chapter 15</b>	<b>Building the Media Player . . . . .</b>	<b>319</b>
	15.1 The Media Player . . . . .	319
	15.2 CXMyMediaPlayerApp — The Media Player . . . . .	320
	15.2.1 Creating the Media Player . . . . .	323
	15.2.2 Loading Controls . . . . .	323

## Contents

15.2.3	The Window . . . . .	324
15.2.4	The List Box. . . . .	326
15.2.5	The Text Box . . . . .	327
15.3	Buttons Are Connectivity. . . . .	327
15.3.1	CXSearchButton — The Search Button . . . . .	328
15.3.2	CXSearchButton — Loading Images . . . . .	329
15.3.3	CXSearchButton — Handling Mouse Clicks . . . . .	329
15.3.4	CXPlayButton — The Play Button . . . . .	331
15.3.5	CXPlayButton — Handling Button Clicks . . . . .	332
15.3.6	CXStopButton — The Stop Button. . . . .	332
15.3.7	CXCloseButton — The Close Button. . . . .	333
15.4	CXMyMediaPlayerApp — Loading Buttons . . . . .	334
15.4.1	CXMyMediaPlayerApp — Cleaning Up . . . . .	335
15.5	Sample Program — Plugging in the Media Player . . . . .	335
15.6	Conclusion. . . . .	337
	Afterword . . . . .	339
Appendix	<b>Recommended Reading</b> . . . . .	341
Index . . . . .		343

# Acknowledgements

There are a number of people who, in one way or another, directly or indirectly, have helped my book through to completion or have ensured the quality of its contents. I would like to take this opportunity to express my gratitude to each of them. My thanks go out to:

**Beth Kohler, Wes Beckwith, Jim Hill**, and all the others of Wordware Publishing, for being efficient people to work with.

**Herb Marselas**, for keeping me and my work in check with his valuable technical revisions.

I would also like to thank my family and friends for their enduring support, advice, understanding... and for everything else they do.

Oh, and one more. I would like to thank you, the reader, for taking the time to purchase my book in the hope of bettering yourself by learning interface development in DirectX.

Alan Thorn. London, UK. 2003.

[DirectX\\_User\\_Interfaces@hotmail.com](mailto:DirectX_User_Interfaces@hotmail.com)

This page intentionally left blank.

# Introduction

At some point through the ages an erudite scholar of great eminence declared that computers are intended to make our lives simpler, they do not make mistakes, and they cannot think for themselves. Like an oral tradition or antediluvian echo, these axioms and countless more have reached my ears long after passing through those of previous computer professionals who, like me, had reservations about their provenance and questions about their accuracy. Those who've labored more than one sleepless night hunched awkwardly over a keyboard, gazing into the cold depths of a lifeless monitor and looking for the one stupid reason why their code won't compile, will no doubt concede that sometimes, just sometimes, these electronic "prodigies" can be far more trouble than they're actually worth. Whether it's a CD drive that won't eject despite the number of times you order it to do so or an application you've told to close but which never does, no number of obscenities you assail the computer with ever seem enough to subdue it into submission. They've given humanity far more than they've ever bargained for, in more ways than one. The scholar's words are wrong, surely?

It needn't be like this, though. The relationship between humankind and machine doesn't have to be a hopeless equilibrium of friend or foe, good or bad, black or white. So how can we reconcile the distinction of human master and computer slave on the one hand with the blatant computer disobedience we've seen on the other? To ask whether it's the fault of one or the other no longer seems productive, and this in itself might mean we're asking the wrong questions. Not knowing how to save a word-processed document or pressing a button that should never have been clickable at all can be attributable to neither mankind nor machine but rather

the communication, or conceptual mediator, between them both, which is known to us as the user interface.

Such communication tragedies have been the Achilles' heel of many an application. So long as there are developers who underestimate the significance of user interfaces, there will be unpredictable programs, often which lead to frustration. Such importance have user interfaces garnered over the past decade, such giant technological leaps has mankind made, and so masterful with machines have we become that it is no longer acceptable for developers to say they have compromised user interfaces in favor of refining the underpinnings. For if the user interface is the only means by which we can access these workings, then surely one becomes just as important as the other.

This book is about how to develop good, solid interfaces. It is both a reference and guide. It is about making the scholar's words a realization. It is about learning that user interfaces are an important step along the path to harmonious union between user and machine. Perhaps the wise scholar's words aren't so dubious after all, provided we understand that its truth rests on how much we make it so. Granted, interfaces are not the answer to all our problems. Even the most excellent of interfaces can't make a bad program good, but a good interface *can* make an equally good program great — this is something we'd do well not to forget, and it is with this understanding that we should approach our tasks ahead...

## Who Should Read This Book

---

With all books there is a “target” audience, and although the distinction between those suited and unsuited to this title is somewhat nebulous, I feel confident in advocating that readers wanting to learn more about DirectX and sensible user interface design, and who have a solid comprehension of Microsoft Visual C++, as well as a basic grasp of general COM principles, are likely to receive most of the benefits this title offers. Knowledge and experience of other related software or technologies can be seen as a bonus for, but not essential to, understanding the examples and

methodologies presented throughout this text. Having said all of that, however, far be it from me to set any pretensions and strict criteria. I wouldn't wish to deter anyone who is keen to learn and willing to try; all are welcome here.

## **What This Book Covers**

---

From concept to completion I illustrate the design, implementation, and testing of a full-formed user interface using DirectX, providing suggestions and rationales for good planning and sound coding. Such important mechanisms as message handling and hierarchical class libraries are detailed with thoroughness, alongside exciting technologies like Direct3D, DirectInput, and DirectShow. At numerous stages throughout this book digressions are made to highlight and detail valuable ideas, from advanced memory management to wrapper classes. Overall, this tome constitutes a good resource for any reader wanting to learn DirectX to create cutting-edge graphics, understand the comprehensive stages of user interface development, learn new tips and tricks, and harness the time-saving qualities of the reusable suite of visual controls, which are provided on the companion CD and coded within the tutorials presented throughout this book.

## **How to Read This Book**

---

In a book dealing with technology and techniques, such as this, it can often be difficult for individual readers with diverse abilities and experience to know what to read, what not to read, where to start, and where to end. Fortunately, this book can serve as both a tutorial and a reference. It represents a complete workflow from start to finish, and is divided into three distinct parts reflecting the foundations of interface design, the building of a UI library, and the implementation of a media player. These are further subdivided into individual chapters, detailing particular development processes and highlighting specific implementation issues. There is no race to the

finish line with this book and there are no time-critical sections, so you can take this at your own pace, going from one chapter to the next. For those new to the topic of user interfaces or DirectX, I recommend reading through this book from cover to cover and working on practical code as advised. Conversely, there are no rules governing what should or should not be read for those already familiar with DirectX or user interface design, or both. It rather depends on an individual's needs and knowledge. For such cases, I suggest viewing the contents and index or flipping through the pages and browsing the headers to find specific areas of interest.

## Technical Requirements

---

To follow the examples and compile the companion CD's code you'll need:

- Windows 98 or above
- Visual C++ 6.0 or above
- DirectX 9.0 SDK (provided on the CD)

It is worth noting, however, that most action-packed or fast-paced computer games developed with DirectX these days list a range of 3D accelerated graphics cards, any one of which is a requirement. While the code and examples presented throughout this book and on the CD do not necessitate such hardware, I would heartily recommend you purchase one of these graphical marvels in the interests of speed, reliability and, above all else, your sanity.

## Companion CD Contents

---

The companion CD is organized as follows:

- Binary and source versions of the examples presented throughout the book are structured into folders using the following convention:

Book Code\`<PART NAME>`\\`<CHAPTER>`\

- As well as spanning chapter folders, the user interface components that are produced in later chapters are collectively included on the CD at: Book Code\`Part II\UI LIB FINAL`

Additionally, I have included the DirectX 9.0 SDK and an evaluation version of Paint Shop Pro on the CD.

## Conventions Used Throughout the Book and CD

---

The first occurrences of new, important words appear in italics. Hints, notations, and general “Did you know” facts are included as inserts above or below relevant passages, while figures and diagrams occupy capacious areas on their own. However, as part of a strenuous effort to accentuate the overall clarity of my C++ code and increase its amenability to all levels of readers when presenting examples, I have employed a composite, and sometimes unconventional, formatting style that I believe takes the best of all worlds. Part of this incorporates not least the infamous nomenclature of Hungarian notation, whose nuances for variable prefixes are explained in a later chapter and which is used wherever possible to enhance code presentation and accessibility. Additionally, sensible indentation and spacious arrangement of code to improve legibility on the printed page and among the source files is used throughout the book and on the CD in my own eccentric flavor with variables given clear, meaningful titles that are both illustrative of their purpose and data type. And finally, in a rather twisted logic, I have chosen to keep code comments on the CD short, blunt, and to a minimum with the intention of actually maintaining a self-documenting nature — I will not overcrowd lines or clutter screen space with needless annotations amounting to a comprehensive novel on what a certain function does or doesn’t do, especially when it is described sufficiently within this book; instead I will often just include relevant chapter references as the comment.

This page intentionally left blank.

---

# User Interfaces

There are a great number of books available that discuss user interfaces and DirectX individually, but there are only a few that address these issues together. Hence, it's primarily for this reason that I've chosen to pen this title. This book is structured into three parts: The first discusses foundational knowledge that underpins both interfaces and DirectX as a whole; the second puts this knowledge into practice by developing a reusable interface library; and the third consummates our efforts as we build a small media player program in DirectX, complete with a user interface. This chapter begins our voyage by examining user interfaces in general; specifically it aims to answer the following questions:

- What is a user interface?
- What is a control?
- What types of controls are there?
- What are interface flow diagrams?
- What is interface design?
- Why is interface design important?
- Which interface design guidelines are recommended to follow?

## 1.1 User Interfaces — What Are They?

---

Conceptually, a user interface is a surface forming a common border between two regions of an application; on one side of the border is the user and on the other side is a computer program. The only way one communicates with the other is through the border. Practically, a user interface consists of gadgets and gizmos, like the buttons and check boxes inside a window that the user manipulates to communicate with a program. For example, a user informs a program of his name by typing letters into a text box, and a program might tell us about an error through a message box or a flashing icon. Because of this presentational role, an interface is sometimes termed the *front end* of an application, and the program itself is correspondingly named the *back end*. Take a look at Figure 1.1 to visualize an interface and its place in an application.

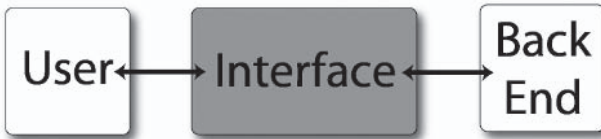


Figure 1.1

Figures 1.2, 1.3, and 1.4 provide some examples of real-life interfaces.

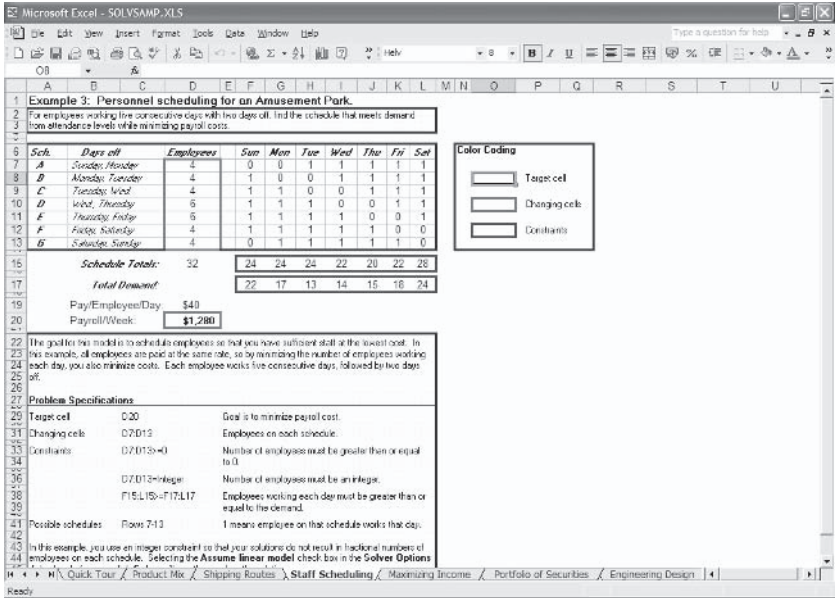


Figure 1.2. Interface of Microsoft Excel from Office XP. This uses toolbars and buttons to provide quick access to program options. User data is entered into a giant grid.

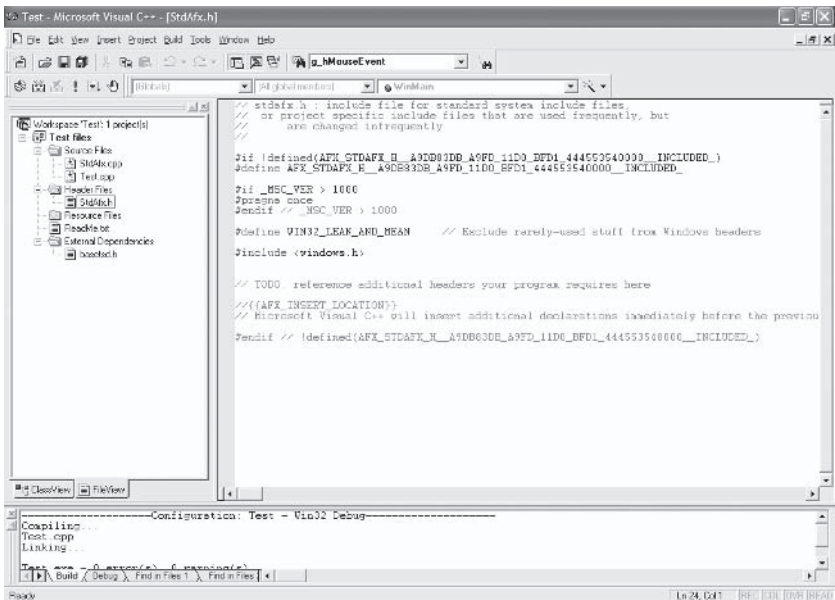


Figure 1.3. Interface of Microsoft Visual C++. A tree of items on the left allows programmers to navigate files, classes, and resources. The buttons and menus provide access to program options, and the debug text pane at the bottom narrates the compilation and linking processes.

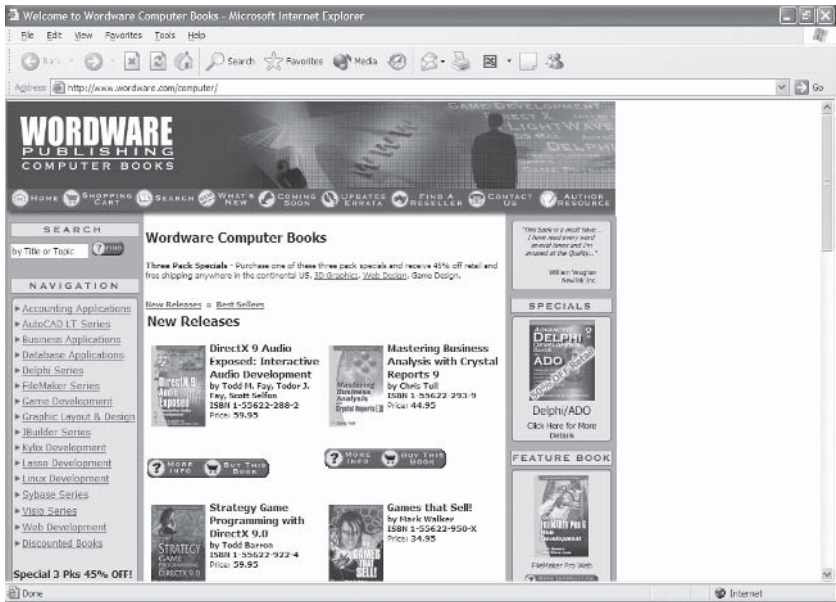


Figure 1.4. Interface of Microsoft Internet Explorer. This allows users to navigate the Internet using the toolbars at the top. You can see different web pages inside the large box.



## NOTE

Sometimes, the terms “interface,” “user interface,” and other variations like “graphical user interface” (GUI) are used to mean distinctive things. However, for the purposes of this book, these names can be used interchangeably.

## 1.2 Controls — Gadgets and Gizmos

It was said previously that interfaces are a collection of gadgets and gizmos, such as buttons, text boxes, check boxes, lists, menus, and more. These are more technically known as controls, or sometimes components, and are referred to as such throughout the rest of this book. These allow the user to talk to a program and allow the program to talk back. As we shall see, there are a great many controls — so many, in fact, that we do not have sufficient space to cover

them all. Instead, the most common types are described in the following subsections, and each of them is investigated more thoroughly as we come to develop them in DirectX later in this book.

## 1.2.1 Text Boxes

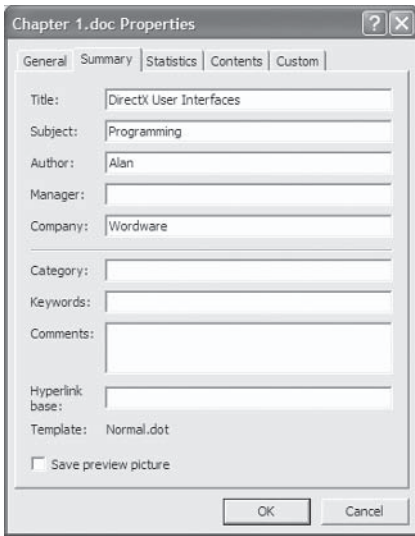


Figure 1.5. Text box controls

Text boxes are rectangular spaces that accumulate characters as a user types. Often they are used to store names, addresses, telephone numbers, and similar information. Indeed, text boxes are an ideal solution for storing text that spans only one line and is from 0 to 255 characters in length. For longer text where paragraphing and arrangement matters, developers should use text edit or rich text edit controls.

## 1.2.2 Text Edits

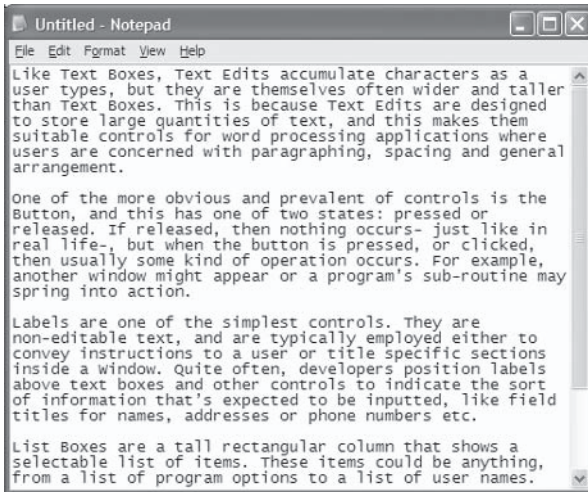


Figure 1.6. Text edit controls

Like text boxes, text edits accumulate characters as a user types, but they are often wider and taller than text boxes. This is because text edits are designed to store large quantities of text, and this makes them suitable controls for word processing applications where users are concerned with paragraphing, spacing, and general arrangement.

## 1.2.3 Buttons

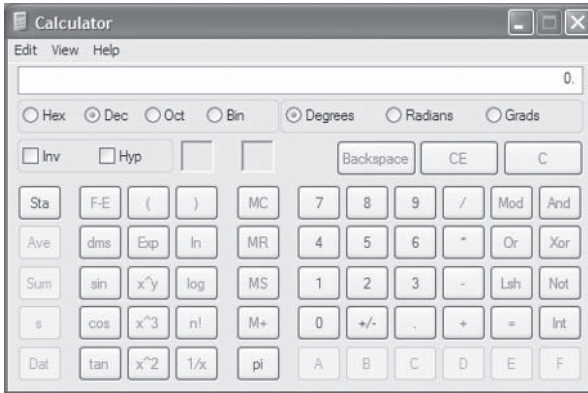


Figure 1.7. Button controls

One of the more obvious and prevalent controls is the button. This control can have one of two states: pressed or released. If released, then nothing occurs, but when the button is pressed, or clicked, then usually some kind of operation occurs. For example, another window might appear or a program's subroutine may spring into action.

## 1.2.4 Labels

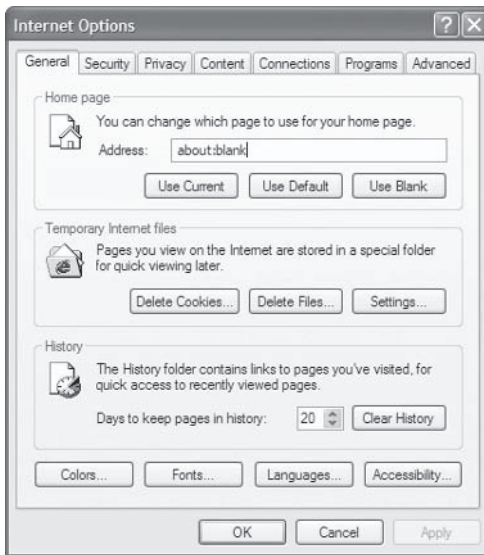


Figure 1.8. Label controls

Labels are one of the simplest controls. They are noneditable text, and are typically employed either to convey instructions to a user or title specific sections inside a window. Quite often, developers position labels above text boxes and other controls to indicate the sort of information that's expected to be input into them, like field titles for names, addresses, or phone numbers.

## 1.2.5 List Boxes

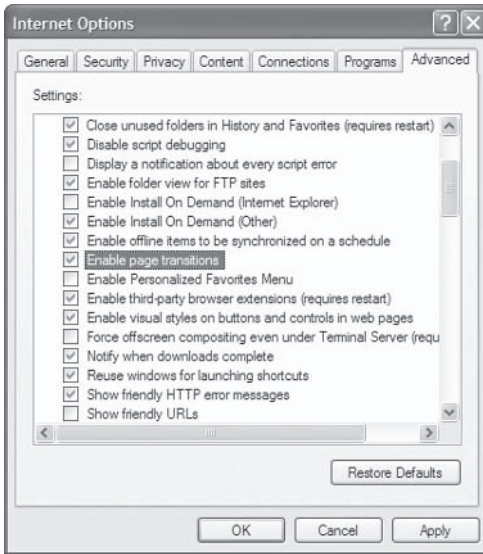


Figure 1.9. List box controls

List boxes are tall rectangular columns that show a selectable list of items. These items could be anything, from a list of program options to a list of user names. List boxes are particularly ideal when a user must either add or remove specific items from a list at run time, such as a company’s employees, or when a user must select a range of different items, like which fields of information should be included in a search. However, in cases where only one item from a list needs to be selected and needs to be visible, drop-down lists typically provide a more convenient solution.

## 1.2.6 Drop-Down Lists

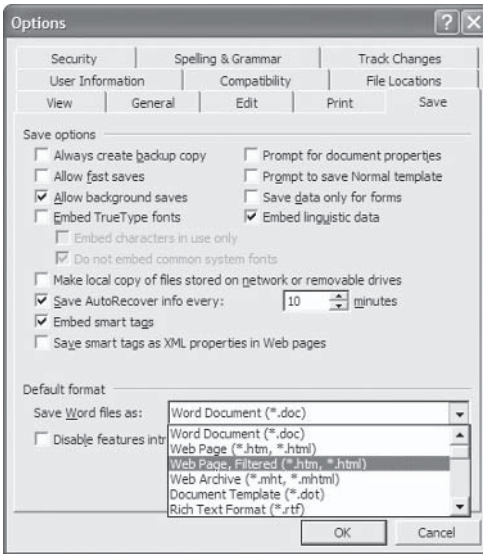


Figure 1.10. Drop-down list controls

At first glance, drop-down lists appear to be like text boxes, except for a downward arrow anchored to their far right-hand side. In reality, however, drop-down lists are more like list boxes, except that only one item can be selected and only the selected item is shown. To preview and select other items in the list, click on the downward arrow.

## 1.2.7 Check Boxes

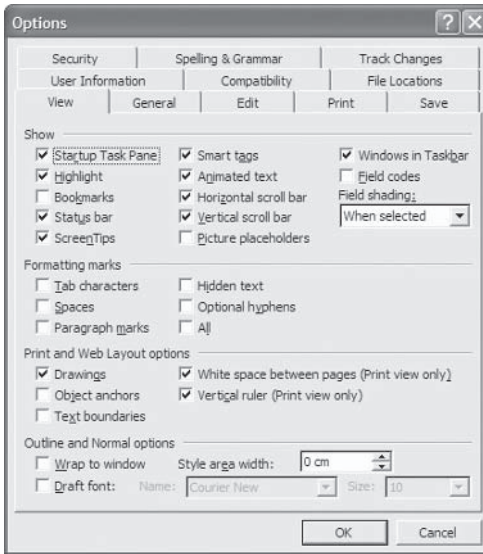


Figure 1.11. Check box controls

Check boxes are essentially label controls paired with a checkable box that can have one of two states, checked or unchecked. Check boxes are useful for a number of different purposes, such as representing data that has true or false values like marital status, employed, or smoker. They also are useful for displaying and setting whether program options are enabled or disabled.

## 1.2.8 Menus

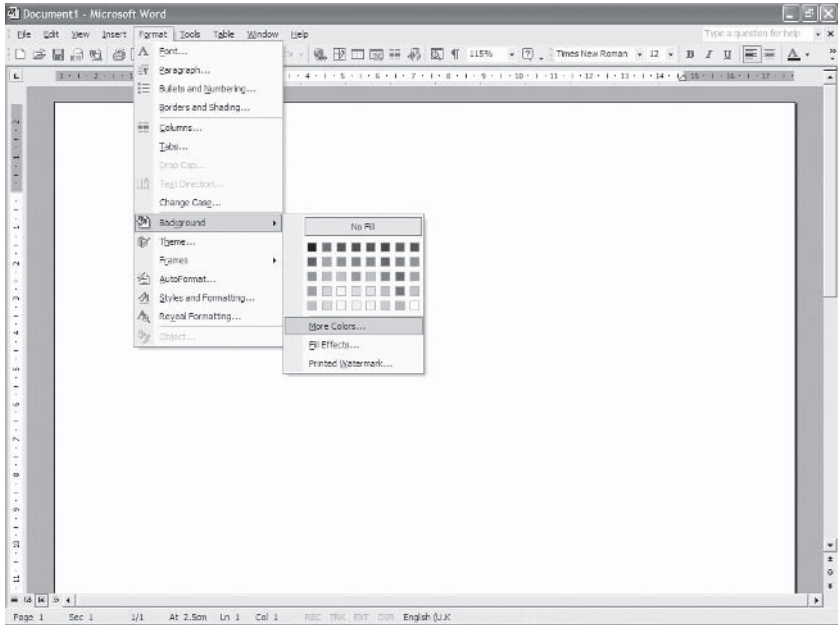


Figure 1.12. Menu controls

Menus are used to display a structured list of program options. Furthermore, they generally take on one of two basic forms, an application menu or a pop-up menu. The former is the standard drop-down menu that appears after clicking on commands like File, Edit, and Tools, while the latter — although it appears and is structured similarly — appears when a user clicks the right-hand mouse button over specific areas inside a window. This is called a pop-up menu.

## 1.2.9 Page Controls/Tab Controls

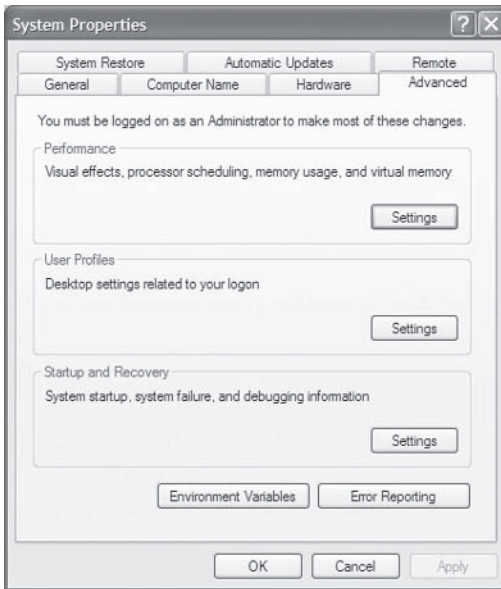


Figure 1.13. Page controls

Page controls are like folder dividers for interfaces, dividing controls into separate screen pages. Their greatest asset to a user is presentation and clearness, and they're most often used to logically divide an interface into related sections. For instance, in a client details application a developer might store customer details on one page, financial information on another, purchase history on another, and so on.

## 1.2.10 Windows and Other Containers

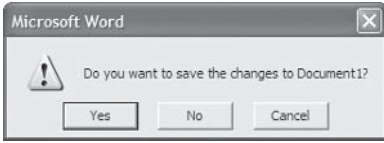


Figure 1.14. A window containing several buttons.

The last form of control to be discussed in this section is the container, of which the most well-known kind is the window control. Essentially, containers are simply controls that contain other controls inside their boundaries. For example, a window can contain numerous buttons, check boxes, lists, and even other containers like windows. Additionally, whenever a container moves or changes size, then all controls contained inside are affected. Because of this important dependence on controls and their containers, the terms *parent controls* and *child controls* are used to designate the containers and the contained, respectively. See Figure 1.14 for an example of a window containing several child controls, and then examine the diagram in Figure 1.15 to see how those child controls are structured hierarchically.

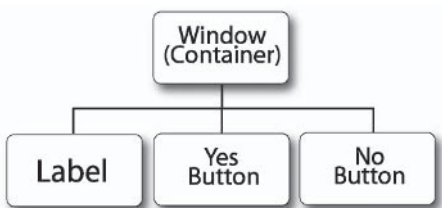


Figure 1.15. Control structure



### NOTE

We develop many of these controls later in this book using DirectX, so there is no need to concern yourself about their implementation at this stage.

## 1.3 Interface Flow Diagrams — Interfaces on Paper

Unfortunately, one of the duller phases of interface development also happens to be one of the most important phases — that of documentation. For example, how should developers explain a program’s interface on paper? How can developers describe to other developers which screens users must navigate in order to perform various program actions? And where might those actions lead? Indeed, because of these considerations it becomes tempting to provide worded descriptions of these processes, but this is not the recommended approach. After all, it has been said that “A picture is worth a thousand words.” Therefore, when documenting an interface, developers should avoid short novels and, instead, strive to represent them in a diagrammatic form. This is where interface flow diagrams come into play. They use boxes and arrows to show a top-level view of an interface. They ultimately show each screen of an application and use arrows to represent the different options at each stage. Take a look at Figure 1.16 to see an interface flow diagram for a hypothetical client details program.

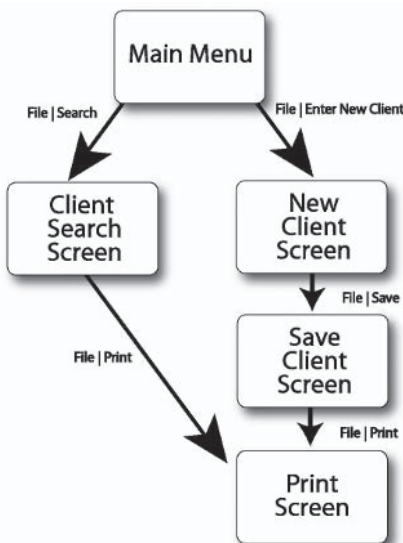


Figure 1.16

**TIP**

Rather than simply using boxes to describe each screen, you might want to opt for screen shots instead. Other good alternatives for previewing the look of an interface are products like Paint, Visio, or PowerPoint. Also, Borland C++ Builder and Delphi offer excellent facilities for designing user interfaces.

**NOTE**

Interface flow diagrams are discussed in Part III of this book where we produce an interface for a media player program.

## 1.4 Interface Design — Tips and Tricks

---

Interface design is about optimizing interfaces to be simpler and more intuitive for users. To an extent, it is feng shui for interfaces. It is about following a series of proven guidelines to make interfaces better. Understandably, at this point you might be asking yourself something like: Why bother with this at all. Can't I just design an interface however I like? Or maybe: Why is interface design so significant that there are companies devoted entirely to this process? Surely, there's no need. However, the answer to all of these questions is surprisingly simple: Users purchase software to get a job done. The easier it is to use, the better. Consequently, the benefits of user-friendly interfaces are manifold: Users make fewer mistakes, companies pay less for staff training, users achieve results faster, your software remains competitive, and your company gains kudos in the industry. The list goes on. In this section we examine various features that are characteristic of successful interfaces. You'll be able to use these as a series of guidelines when developing your own, but this is by no means a comprehensive list. Some of the interface design issues that are addressed in the following subsections include the extent to which artwork can be used in interfaces without having negative implications, the significance of aligning controls in columns or rows, and the importance of handling errors using suitable methods.

 **TIP**

There are various publications available that document Windows interface guidelines thoroughly. See the recommended reading section.

## 1.4.1 **Be Consistent**

One feature above all others gives interfaces solidarity and users peace of mind, and that is consistency in design. Humans find comfort in the familiar, so if users can navigate list box controls using the keyboard arrow keys in one screen, they should expect to be able to do so in all screens throughout an application. In fact, an endless list of consistencies should be applied to other controls too. For example, interfaces for Western nations should be designed such that they flow from top to bottom and left to right. Consequently, buttons like OK and Cancel — those which complete operations or represent a change in program flow — are ideally positioned to the right or the bottom of the screen, or even the bottom right. Pop-up menus should appear only when a user clicks the rightmost mouse button or presses an appropriate context key on the keyboard.

## 1.4.2 **Know Your Audience**

Decisions about whether to include features such as *wizards* in your software will come from understanding your audience. Statistics suggest that novices appreciate the simplicity and brevity that wizards offer, while more advanced users prefer greater control and flexibility at the expense of such simplicity. Therefore, it would be of little value developing a paintbrush application for home users without considering the value of a wizard to achieve complicated tasks. In short, respectable developers won't disregard their customers' requirements any more than an author would ignore his readers. If you even hope to sell your software, then you must develop to consumer demand, so research well and act accordingly.

 **TIP**

Wizards are small applications that allow a user to perform complicated operations in a series of progressive steps. Often, a user moves onto the next step by clicking the Next button. For example, Microsoft Visual C++ often uses wizards to guide you through creating a new project.

### 1.4.3 Justification and Alignment

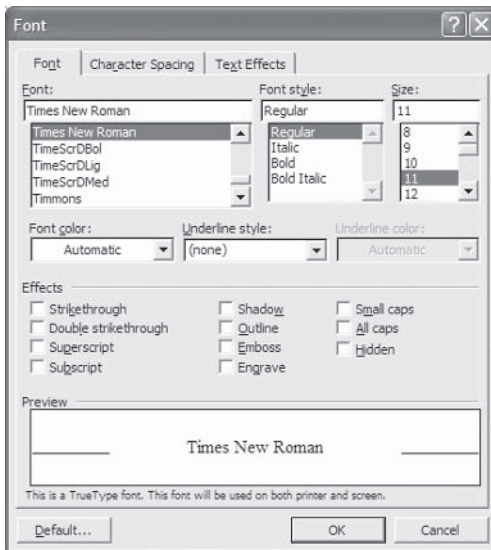


Figure 1.17

Sloppy presentation leads to confusion, and confusion leads to mistakes, and mistakes — as far as users are concerned — are a bad thing. Thus, it is principally because we want to avoid confusion that justification and alignment become important aspects for interface design. It is therefore advisable to follow these broad guidelines when considering the arrangement of controls: Ensure related controls are arranged in harmony with one another, such that they are placed level in rows or columns and that a sufficient distance exists between them. Spacing is an important topic too; too crowded an interface becomes difficult to follow, while too spacious an interface appears amateurish and tiresome. It is important

to remember also that in Western nations, people read from left to right and top to bottom, while in others people read differently — so be sure to arrange interfaces appropriately for your audience.

## 1.4.4 Grouping Data

The human mind tends to put things into logical categories or groupings. Consequently, it makes sense to arrange your interfaces accordingly. For example, if you were developing a database to store client details, it would seem reasonable to group related fields together in close proximity on screen; name, address, and telephone in one group and payment details, credit history, and bank account in another. Page controls provide an excellent solution to this problem. See Section 1.2.9.

## 1.4.5 Error Handling

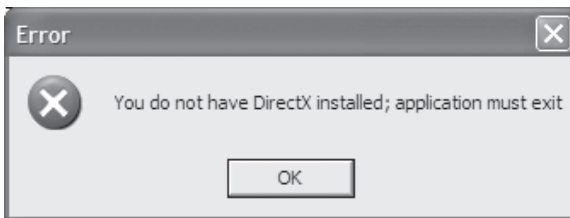


Figure 1.18. Notifying a user of an error

For some reason or other, it seems impossible to eradicate all errors from a program, and so it must be accepted as a fact that at some point an error might occur. The general rule of thumb is to handle errors gracefully, so be certain to notify a user of an error through a message box and then exit. Don't simply quit a program without warning and, even more importantly, don't allow a user to continue if a fatal error does occur.

## 1.4.6 Disabling Program Features

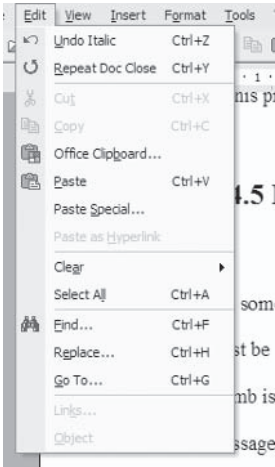


Figure 1.19. Disabled features

Every so often, a specific program feature might become unavailable because it currently serves no purpose, such as the spell checker on a blank word processing document or the delete operation on a write-protected file. Traditionally, the way interface designers handled this problem was to hide the buttons or menu items that triggered these operations until such time as they could be used. However, such a practice has served more to confuse rather than help users since it would seem that one moment the button was there and the next it wasn't. Now developers choose to *disable* unavailable features instead. They do this by graying out the buttons and menu items instead of hiding them. During this time those features cannot be clicked or invoked in any way, but at least the user is still aware of their existence and will understand why they have been disabled. In short, gray out unavailable features; do not hide or remove them.

### 1.4.7 Graphics, Colors, Icons, and Art

I'm not going to point fingers and offer recriminations, but the use — or rather, overuse — of artwork in interfaces is where a great number of applications fall down, even today. Primarily, this is because it's deliciously tempting to be artistic, and particularly so in computer games and with DirectX. However, art, and graphics in general, must be used sparingly and appropriately — not just for the sake of it. Colors are good for representing application states — green for success and red for error, etc. But even so, colors alone cannot be relied upon since many users may be color blind or visually impaired. Pictures in general can make an interface look charming, giving it much needed character and individuality, but again, overuse will confuse your users and cloud exactly what it is that the program is supposed to be achieving. Finally, as we see in the next section, icons are an ideal solution for cutting down on text; they convey a message and make interfaces look good at the same time. But even here there is a rule of consistency to be acknowledged: Do not use the same icon to mean two different things in the same application. For example, if a button has a folder icon to represent a File | Open command, don't use the same icon elsewhere to run a Search operation.

### 1.4.8 Balancing Text and Symbols



Figure 1.20. Toolbar icons

For the author among developers, it will no doubt be inviting to write on-screen instructions to users of your interface, like “Please enter your name.” However, text should be used as sparingly and as concisely as possible in an application, partly because users dislike reading and partly because it reduces translation costs for developers. Therefore, any one element of text should not exceed two sentences in general. In cases where it's longer or where a single verb will suffice, it is often advisable to use icons as a symbol

instead, within the guidelines of the previous section, of course. Symbols are images that suggest something else, like an action or a process. For example, the File | Open command is often associated with an icon depicting a folder and an outward-pointing arrow, and the File | Save command is paired with an icon of a disk. So, the thrust of this section is: Keep text short and use images descriptively.

## 1.4.9 Paths and Navigation

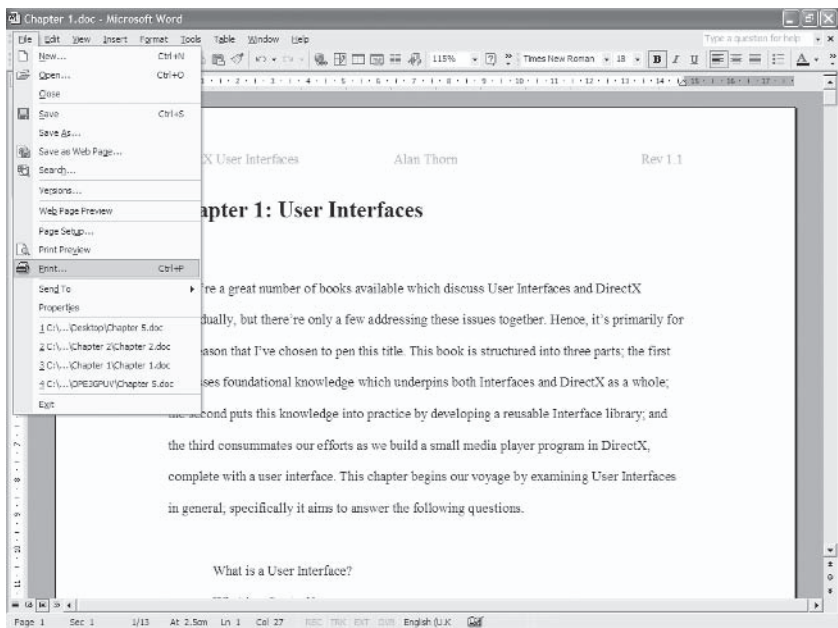


Figure 1.21. In Microsoft Word printing requires two steps. The first step is to choose *Print* from the *File* menu.

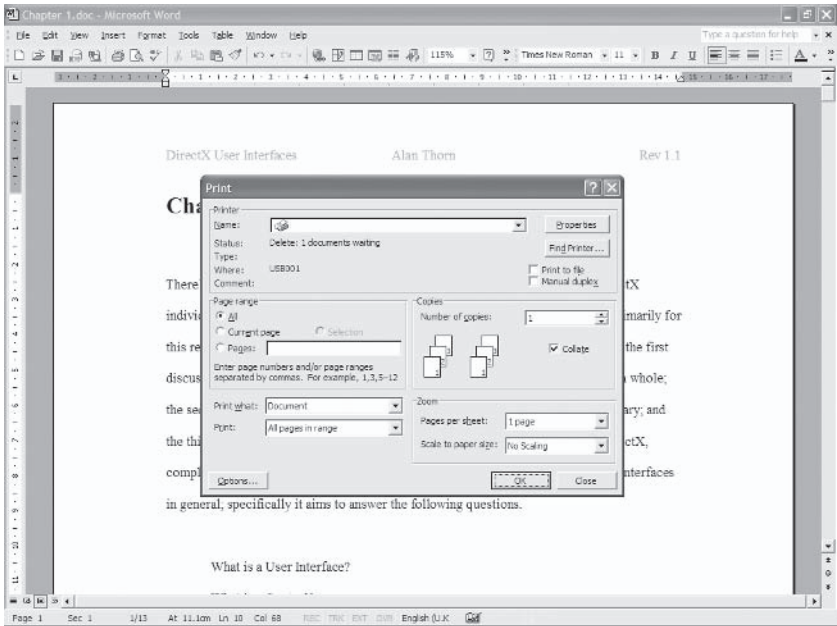


Figure 1.22. The second step is to click OK on the Print window.

When people get together and discuss how to achieve things using your software, it’s quite likely they’ll quote which route to travel through your interface. They’ll say things like “Click on File | Edit, then Options and then...” and so on. However, if people aren’t sure which steps to take to reach various screens or, even worse, if people aren’t sure how to exit certain screens to get back whence they came, then your interface is not providing a clearly traceable, or retraceable, path. To help prevent such problems from occurring, always be sure your interface flows logically, and be certain that users only perform the necessary actions when navigating from screen to screen. Always be sure that screens can be exited, and that destructive operations are not performed without user confirmation and acknowledgment.

## 1.4.10 Keyboard Support

No matter how much of a favor developers might think they're doing users by limiting keyboard use and making everything mouse oriented, there are distinct drawbacks to supporting mice at the expense of keyboards. You should note that advanced users and other professionals, such as telephone operators, often prefer keyboard shortcuts because they're faster and more propitious for time-critical tasks. Additionally, some people find mice difficult to use. Indeed, interfaces are by nature visual entities and therefore mice are likely to be a more natural choice for input, but why do so at the *expense* of keyboards when you can do so as a complement?

## 1.4.11 Tool Tips

A great way to make your software simpler and more intuitive for average users is to provide them with constant but discreet help and advice. This is where tool tips come into play. Tool tips are a form of context-sensitive help where small labels or bubbles of advice appear and vanish as a user moves the cursor over controls.

## 1.5 Conclusion

---

This chapter presented a broad outline of interfaces and the controls that comprise them. It has not been too technical and has not investigated the specifics of each component. This occurs later in the book. Before moving on to examine how interfaces are created using DirectX, let's recap briefly what we've learned in this chapter.

- Interfaces are a surface forming a common border between a user and a program. They are made from a collection of gadgets and gizmos, called controls.
- There are many different sorts of controls: buttons, list boxes, check boxes, labels, page controls, windows, and more.

- The design of an interface can be expressed in an interface flow diagram. This uses screen shots, boxes, and arrows to illustrate how users can navigate an interface and the consequences resulting from their actions.
- Interface design is a time-consuming process. It involves optimizing an interface to be simpler and cleaner to use. This involves adhering to various standards, such as limiting the use of text and graphics.

This page intentionally left blank.

---

# Introducing DirectX

The previous chapter detailed at length both what user interfaces are and their significance in applications, but it omitted a discussion of what tools are at our disposal for developing them. We address this topic as we examine DirectX, a subject which begins in this chapter and continues in stages throughout the remainder of Part I. This chapter is dedicated to answering the following questions about DirectX:

- What is DirectX?
- What APIs does it include?
- What other features does DirectX include?
- Why use it?
- From where is DirectX obtained?
- What are its minimum requirements?
- What's the difference between Debug and Retail mode?
- How is DirectX installed?
- How can Visual C++ be configured to use DirectX?

## 2.1 DirectX — What Is It?

If you've ever wanted to produce state-of-the-art computer games for the Windows platform — complete with fast-paced graphics and 3D sound — then DirectX is undoubtedly *the* technology for you. Simply put, DirectX is a suite of interdependent APIs, or libraries, that provide an exciting feature set to developers. This includes features to create and display 2D/3D worlds, play sounds and music, play videos and movies, create multiplayer games, and read data from a range of input peripherals. Another benefit of DirectX is that developers rarely need to worry about what hardware the user has. This is because DirectX achieves device independence; it acts like a bridge between software and hardware. In other words, developers needn't program hardware directly. Instead, they program exclusively with DirectX, and DirectX in turn takes care of the hard work for us. Thus, so long as hardware manufacturers ensure their products — like sound cards and graphics cards — are DirectX compliant, then DirectX software will work transparently with all such hardware. The following subsections summarize each of the APIs that DirectX is made from (see Figure 2.1).

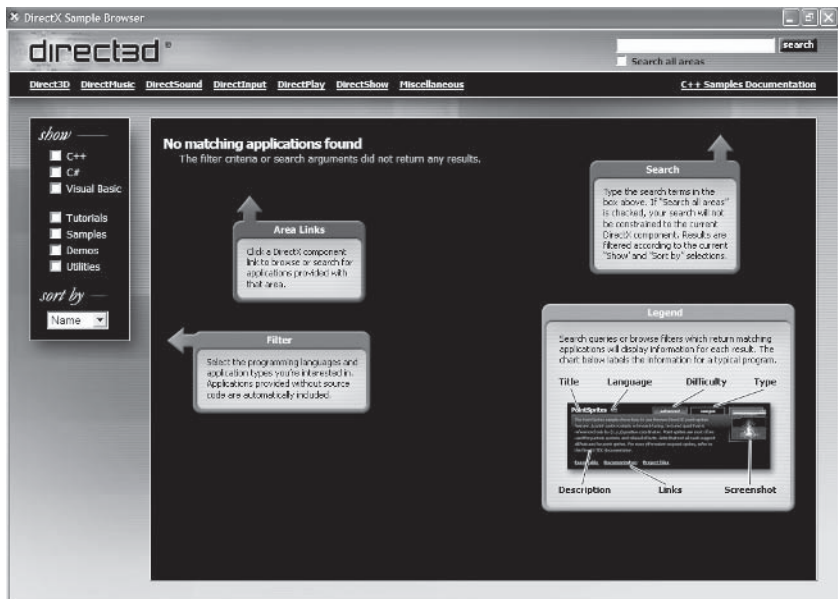


Figure 2.1

## 2.1.1 Direct3D — Graphics

Direct3D is arguably the most well-known and complicated of all DirectX components. Its primary purpose is to present images on screen, both 2D images — like those loaded from bitmaps and JPEGs — and 3D images — like 3D models exported from modeling software. Furthermore, Direct3D utilizes hardware acceleration where present on a user’s graphics card, which means images are typically drawn at incredible speeds. For the purposes of this book, Direct3D is employed to draw user interfaces on screen. We’ll examine how to achieve this in Chapter 3.



---

### NOTE

To use Direct3D in your C++ projects, you must include the relevant header files and link to the appropriate library files. For Direct3D, these files are `d3d9.h`, `d3d9.lib`, `d3dx9.h`, and `d3dx9.lib`.

## 2.1.2 DirectInput — Keyboards, Mice, and Joysticks

DirectInput is aptly named because it allows developers to read data from input peripherals, namely keyboards, mice, and joysticks. The term “joystick” refers to standard joysticks as well as game pads and even wheel and pedal devices. For user interfaces though, it will be sufficient to read data from keyboards and mice only. (See Chapter 4.)



---

### NOTE

To use DirectInput in your C++ projects, you must include the relevant header files and link to the appropriate library files. For DirectInput, these files are `dinput.h` and `dinput.lib`.

### 2.1.3 **DirectMusic and DirectSound — MIDI and WAV**

DirectMusic and DirectSound are complementary APIs, of which the former plays MIDI files and the latter plays WAV files. Collectively, these APIs have sometimes been termed “DirectAudio.” Regardless of which names you use, both APIs have the ability to play sounds as well as a number of different effects, such as echoes and even 3D effects that add a sense of depth and position to enhance the realism of your products. Unfortunately, as powerful and awe-inspiring as these two APIs are, we won’t have a need to use them in our interface work in this book. To learn more about these, I recommend viewing both the DirectX SDK documentation and selected titles listed in the appendix.

### 2.1.4 **DirectPlay — Networking**

DirectPlay allows developers to create multiplayer games. Essentially, you can manage connections over two or more computers and send data between them. Again, as exciting as this component might be, we will not have cause to use it within this book. For more information, however, you can refer to the SDK documentation and the appendix.

### 2.1.5 **DirectShow — Programmable Media Player**

Last but not least there is DirectShow, a programmable media player that plays streamable media, such as MP3s and MPEGs. The mechanism it adopts to play these files can be thought of as a graph, called a *filter graph*, containing several different procedure blocks that are connected to one another by pins. As file playback begins, data travels from a starting point on the graph, and then passes through the pins and into different blocks. There, each block processes and interprets the data appropriately. Then finally, as data exits the graph it becomes comprehensible information, which can be played back meaningfully to a user. In Part III of this book we investigate DirectShow, and specifically how to play media files, when we create the interface for a working media player.



**NOTE**

To use DirectShow in your C++ projects, you must include the relevant header files and link to the appropriate library files. For DirectShow, these files are Dshow.h, Strmiids.lib, and Quartz.lib.

## 2.2 DirectX — Other Features

Even though developers typically choose DirectX because of the above-listed APIs, there are still a number of other utilities and applications bundled with the SDK that are designed to make a developer's life easier. This section now explores some of these.

### 2.2.1 Mesh Viewer

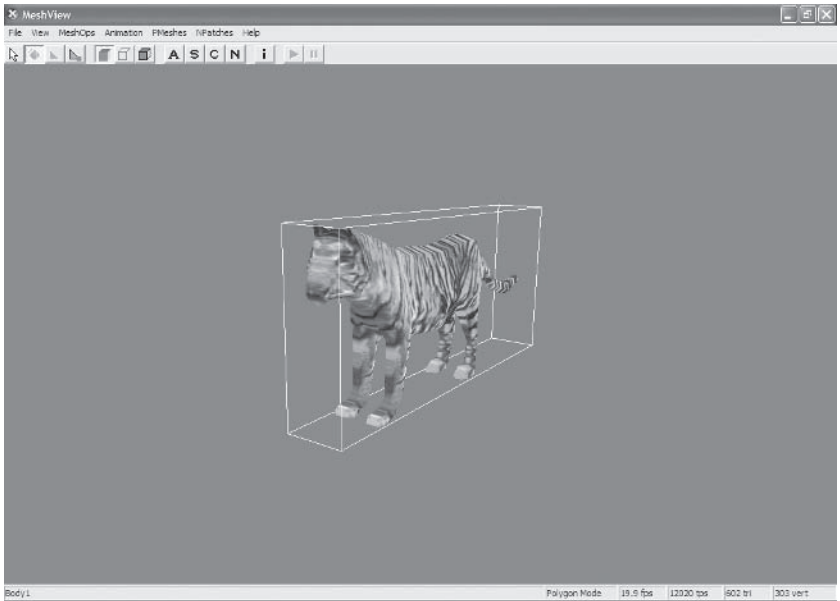


Figure 2.2. DirectX Mesh Viewer

For those who are, or those who know, talented graphic artists proficient in 3D modeling products such as 3D Studio MAX and Maya,

the Mesh Viewer application might prove to be a useful tool. Its purpose is to preview 3D models that have been exported into a format that DirectX can understand. Using this tool, one can estimate how a model will look in an application. You can also view model statistics such as polygon counts, etc.

## 2.2.2 Error Lookup

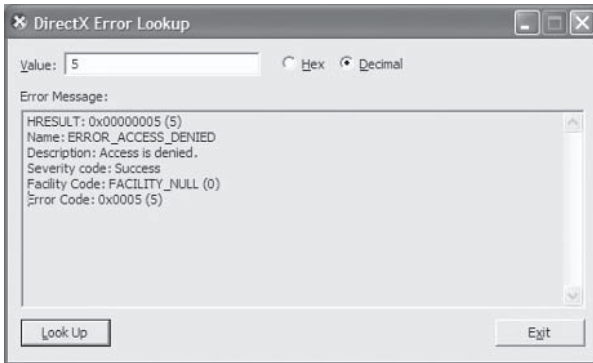


Figure 2.3. *DirectX Error Lookup*

DirectX Error Lookup allows a developer to enter either hex or binary error codes — those returned from DirectX functions — and read more about the error’s meaning. This can help you discover exactly what might have caused the error.

## 2.2.3 Caps Viewer

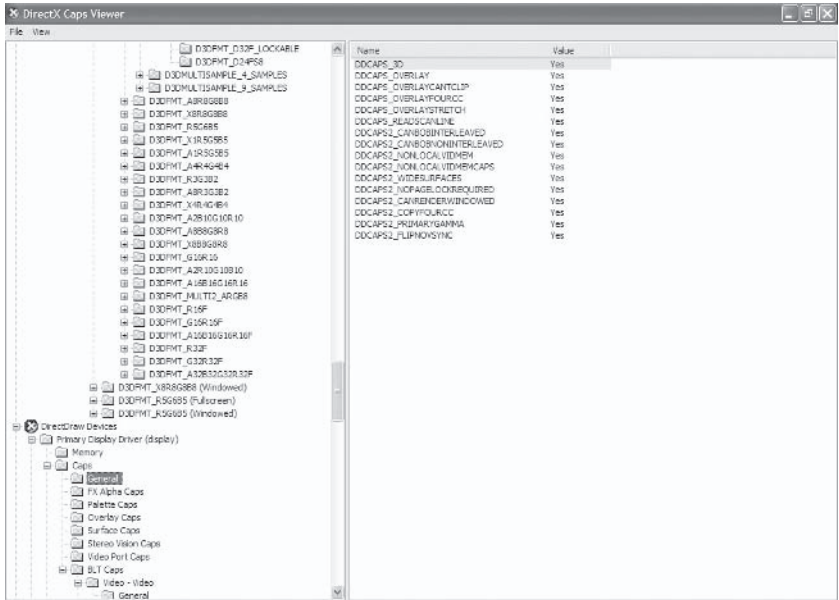


Figure 2.4. DirectX Caps Viewer

“Caps” means “capabilities,” and the DirectX Caps Viewer is an informative tool that lists the capabilities of a computer’s hardware. This information covers a number of different devices, from sound cards to graphics cards, and these categories are selectable from the program’s tree view. Once selected, detailed information is shown in the right-hand pane.

## 2.2.4 GraphEdit

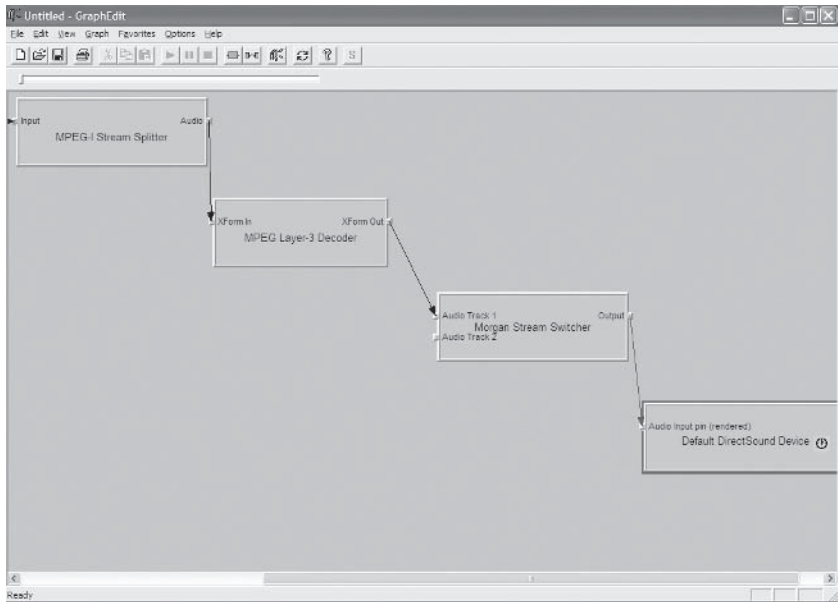


Figure 2.4. GraphEdit

GraphEdit is a DirectShow tool with which users can visually build filter graphs to play streamable media, such as MP3s and MPEGs. As mentioned in Section 2.1.5, a filter graph represents a flow of information; it shows a visual path along which a file is decoded into something meaningful. This is achieved by plotting various procedure blocks onto the graph, and these are connected to one another by pins, like a network of wires.

## 2.2.5 Texture Tool

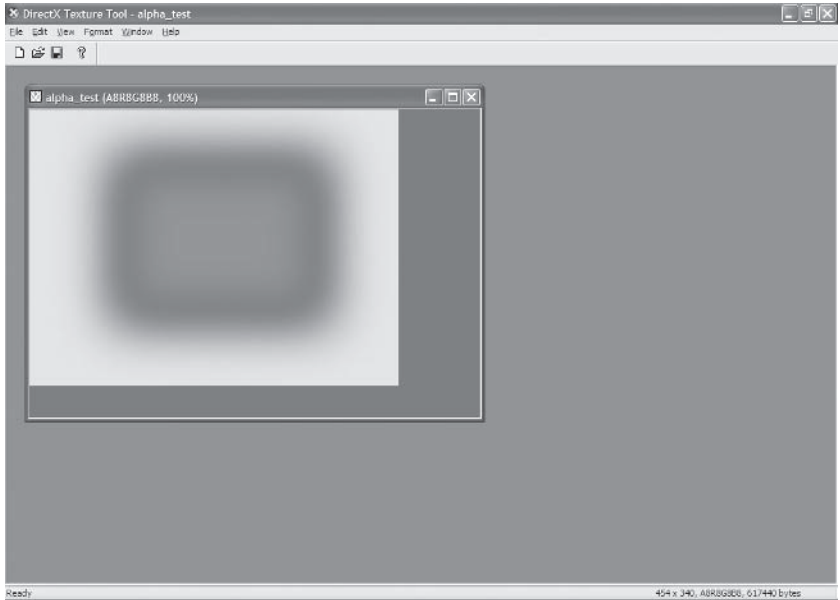


Figure 2.6. Using the Texture Tool

The DirectX Texture Tool creates new textures and edits existing textures that are going to be included inside a 3D environment. Simply put, a *texture* is an image that is projected onto the surface of a 3D model to make it look more realistic. For example, an image of a brick wall can be pasted onto each side of a cube to make it seem as though it were actually manufactured from bricks. It's worth remembering, however, that textures can be created in other image editing programs too, such as Paint Shop Pro or Adobe Photoshop.

## 2.3 System Requirements

---

Before installing the DirectX 9 SDK it is advisable to check your computer's specifications and ensure they meet the following DirectX system requirements:

- Windows 98, Windows Millennium Edition (Windows Me), Windows 2000, or Windows XP
- Approximately 65MB hard disk space
- It is also advisable but not required to have both a 3D accelerated graphics card and a sound card.



---

### NOTE

To determine which hardware is installed in your computer and the particular abilities of these devices, you can use the system information utility, which is usually available from the Windows Start menu. Furthermore, you can also use the DirectX Diagnostic Tool, which can be initiated by clicking Start | Run, typing DXDIAG, and pressing the OK button.

## 2.4 Where to Obtain DirectX

---

Having been introduced to DirectX and probably excited by its capabilities, you're no doubt eager to obtain a copy of the SDK. Fortunately, DirectX is freely available and the latest version (9.0b at the time of writing) can be downloaded from DirectX's home page (<http://www.microsoft.com/windows/directx/>). However, for your convenience, the DirectX 9.0 SDK has also been included on this book's companion CD where it can be found in the DirectX SDK folder.

## 2.5 Installation

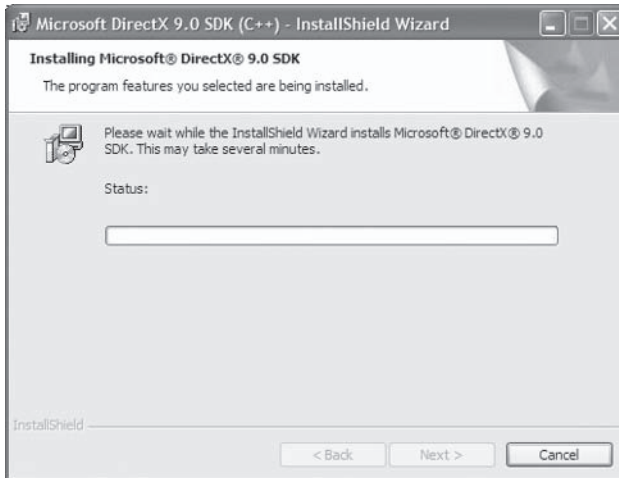


Figure 2.7. Installing the DirectX 9.0 SDK

Installing DirectX is a simple process; just run Setup to begin. It's a standard installation wizard, which requires a path for a destination and provides the usual customization options. Before installation, however, please keep in mind the following points to reduce mistakes. These points have been selected from the readme file and further information can be found there, if required.

- Always uninstall previous versions of the SDK prior to installing newer versions. To uninstall previous versions, click on Control Panel | Add/Remove Programs, and then select DirectX SDK from the list before clicking on the Add/Remove button.
- The DirectX 9.0 SDK cannot be installed on Windows 95, Windows NT, or NEC PC98 systems.
- Installation on Windows 2000 or Windows XP requires administrator privileges.
- A number of virus protection programs can disrupt the installation. For this reason, they should be temporarily disabled until completion.

**NOTE**

During installation you may be given the choice to install DirectX in one of two modes. One is Debug mode, which provides extra error information that developers can use to make debugging applications easier, and the other is Retail mode, which generally provides less support but runs faster than Debug mode. Most consumers will be running Retail mode, but for developers new to DirectX it is recommended they use Debug mode to assist in learning.

## 2.6 Installed Files

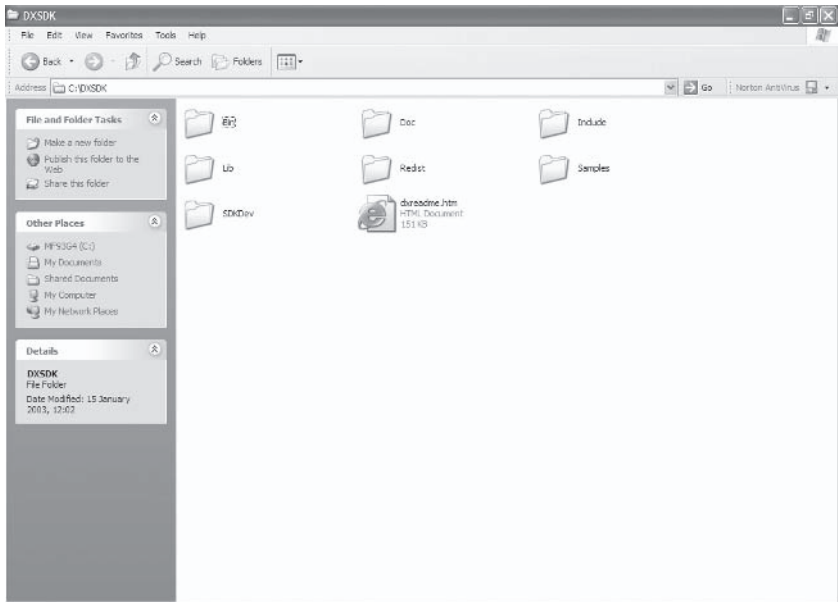


Figure 2.8

After DirectX has successfully been installed, several entries will have been created on the Windows Programs menu. Also, a number of files and folders will have been copied to the installation target. These folders, and their purpose, are listed and explained in Table 2.1.

**Table 2.1. DirectX folders**

<b>Bin</b>	Contains a large number of miscellaneous files and programs, many of which are accessible from the Windows Programs menu. These include the DirectX Texture Tool and other utilities listed above.
<b>Doc</b>	Houses all DirectX developer documentation. You'll make good use of these throughout the course of this book.
<b>Include</b>	Contains C++ source and header files from the DirectX SDK. Some of these are included in our DirectX projects.
<b>Lib</b>	Like Include, Lib contains all respective libraries in the DirectX SDK. Again, some of these are linked to in our DirectX projects.
<b>Redist</b>	Redist means redistributable. This folder contains those DirectX files that can be redistributed legally to users of your products.
<b>Samples</b>	Contains various examples, such as the SDK tutorial projects in Visual C++, etc.
<b>SDKDev</b>	Two separate DirectX installation programs, one to install DirectX in Debug mode and the other to install DirectX in Retail mode.

## 2.7 Configuring Visual C++

At this point, if you started a new VC++ project and included headers or libraries from the newly installed DirectX folder, you'd probably receive a large number of errors at compile time. Primarily, this is because Visual C++ must be told where new libraries and headers exist so it can locate and compile them successfully. To specify these settings in Microsoft Visual C++ and to ensure your projects are DirectX compliant, perform the following steps.

For Microsoft Visual C++ 6.0:

1. Click **Tools | Options** from the main menu.
2. Select the **Directories** tab.
3. Select **Include Files** from the right-hand drop-down list.
4. In the Directories list beneath, add a path to the top of the list that points to the DirectX\Include path. To do this, click on the **New** tool button.

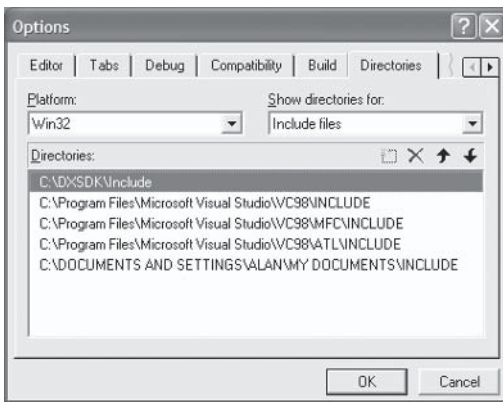


Figure 2.9

5. Then select **Library Files** from the right-hand drop-down list.
6. Again, in the Directories list beneath, add a path to the top of the list. This time, provide the DirectX\Lib path.
7. Click **OK**.

For Microsoft Visual C++ .NET:

1. Click **Tools | Options** from the main menu.
2. Click the **Projects** category.
3. Click **VC++ Directories**.
4. Select **Include Files** from the top-right drop-down list.

5. Add a path to the DirectX\Include folder using the **New** tool button.

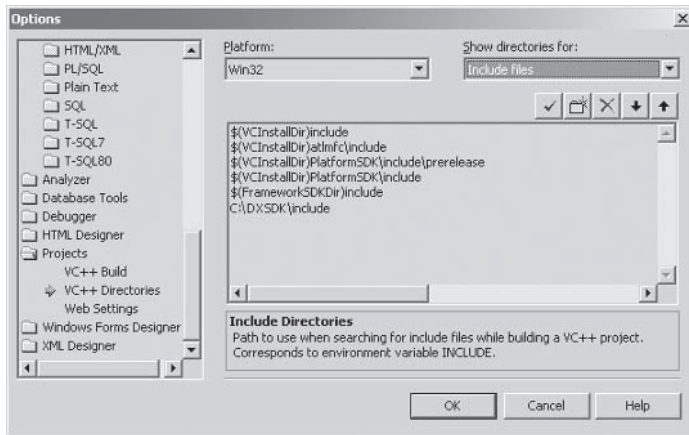


Figure 2.10

6. Select **Library Files** from the top-right drop-down list.
7. Add a path to the DirectX\Lib folder using the **New** tool button.
8. Click **OK**.

**TIP**

You should ensure that the DirectX Include and Lib paths appear at the top of the list. Otherwise, alternative files matching the same name may be used instead.

## 2.8 Coding with Hungarian Notation

---

As a project's code grows in both complexity and size, developers adopt certain coding conventions to make reading and debugging their projects simpler. The most prominent of these is Hungarian notation, a convention that is prevalent throughout the code of both the SDK samples and the SDK documentation. This convention works by prefixing variable names with one or more lowercase letters — sometimes followed by an underscore — and a developer selects these letters to describe information about a variable's data type or relationship. This means that a programmer reading someone else's code doesn't need to search for a variable's declaration to know what sort of data it can reasonably be expected to contain. This is because one can simply determine this information immediately from a variable's prefix. And that, ladies and gentlemen, is Hungarian notation in a nutshell — or a paragraph anyway. Take a look below for some examples of common variable prefixes.

```
DWORD dwCounter;  
char chLetter;  
char* pName;  
int* pNumber;  
int iInteger;
```

---

 **TIP**

It is worth mentioning that Hungarian notation only truly serves its purpose as long as developers adhere to the formatting guidelines. Do not simply select where it will and will not apply. If you're going to use it, then do so properly.

## 2.9 Conclusion

---

For some, this chapter was no doubt a piece of cake. For others, however, it will have provided a valuable insight into how DirectX is installed and configured. Subsequent chapters therefore shall presuppose knowledge of this process, and instead will concentrate on how to use specific components of DirectX, beginning with Direct3D.

This page intentionally left blank.

# Introducing Direct3D

Nowadays, computer games are dispatched to the masses on compact discs, mini discs, or DVDs, and their graphics seem as real as ever. One of the successful technologies that has helped precipitate this graphical evolution is Direct3D. As the previous chapter explained, we use this API to render images to the screen. This chapter presents an introduction to using its features and examines how we can apply them to draw user interfaces. Particularly, this chapter will explain the following points:

- How mathematics is used to express positions and represent transformations
- How a Direct3D application is created and configured in Visual C++
- The programmatic requirements of a Direct3D project
- How images can be rendered to the screen
- How to perform special effects like alpha blending

## 3.1 Direct3D Concepts — Overview and Mathematics

---

Direct3D is an API that represents, manipulates, and draws objects that exist in 3D space. The process of drawing them to the screen is known as *rendering*, or *presenting*, and this occurs much like a camera taking a photograph of a real-life environment. It is a freeze-frame of a 3D world, taken from a particular vantage point and at a specific period in time. In most games or multimedia products, renderings are taken on an extremely frequent basis, like several times a second. This is to ensure that animations can be shown and that an updated view of the 3D environment is maintained as the states of objects change.

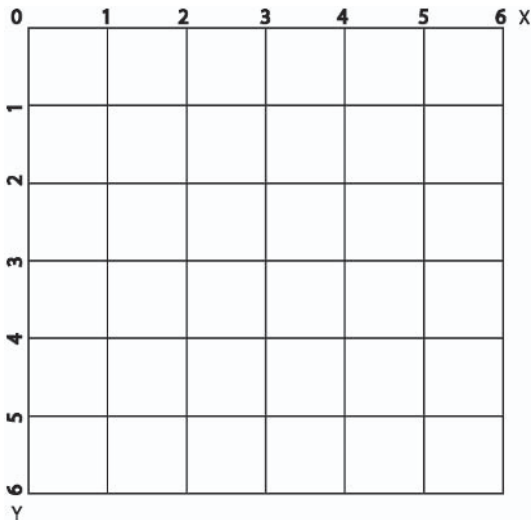


Figure 3.1

Once a rendering has been taken, it becomes visible on-screen. The screen itself has its own 2D coordinate space. It has two axes, X and Y, which stretch across the width and height of the screen, respectively. These typically intersect at the origin in the top-left corner (0,0), and distances along these axes are measured in *pixels*.

Common screen dimensions are 640x480, 800x600, and 1024x768. To express positions in this coordinate space we use a `D3DXVECTOR2` structure. Example positions are (2,5) or (0,7) or (9,0). Figure 3.1 demonstrates the coordinate space of the screen, and the code below demonstrates how we can use `D3DXVECTOR2` to express a 2D position.

```
D3DXVECTOR3 Position;    //Declare position object
Position.x = 5.0f;      //Set X value
Position.y = 7.0f;      //Set Y value
```

 **TIP**

Later sections in this chapter demonstrate further how this structure is used in Direct3D. For those interested, the SDK lists various functions that perform operations on this positional and vector data. These include adding, subtracting, and multiplying them, among others. These functions are `D3DXVec2Add`, `D3DXVec2Subtract`, and `D3DXVec2Scale`.

As I've mentioned, objects rendered by Direct3D exist within 3D space. Such space can be visualized as a giant cube that uses a 3D Cartesian coordinate system. It has three axes that intersect at the cube's center. This is its *origin*. The axes are: X, which extends across the horizon, Y, which stretches vertically, and Z, which reaches into the distance. Measurements to one side of the origin are positive, and measurements to the other are negative. Consequently, we express positions inside 3D space using the conventional (X,Y,Z) notation. See Figure 3.2 to help visualize these concepts. A single point is known as a *vertex*. Example vertices could be: (3,5,7) or (4,5,-3) or (-5,-7,3). Like 2D positions, these values are expressed programmatically using a `D3DXVECTOR3` structure.

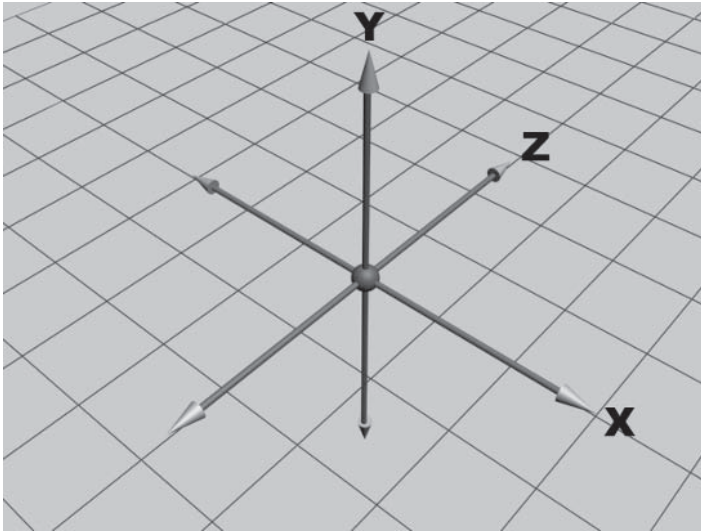


Figure 3.2



#### **NOTE**

Naturally, user interface development will take this book on a more 2D route. We are more interested in drawing windows and controls at various positions on screen than we are with manipulating objects in 3D. Therefore, in-depth discussions of 3D mathematics and 3D-specific features of Direct3D are beyond the scope of this text. For those who are interested, I suggest viewing the recommended reading section in the appendix.

## **3.2 Getting Started**

Direct3D projects can be created in Visual C++ using a number of different methods. The simplest is to create a standard Win32 application, but you could of course opt for other templates. Whichever you choose, it is important to ensure that all projects include the appropriate headers and link to the relevant libraries. These files are `d3d9.h`, `d3dx9.h`, `d3d9.lib`, and `d3dx9.lib`. Apart from these specifics, Direct3D applications essentially begin no differently than standard Windows applications. You must still create a `WinMain` function, still create a window, still create a message loop, and still

process messages. The beginnings of such a Direct3D project can be seen in the following code fragment.

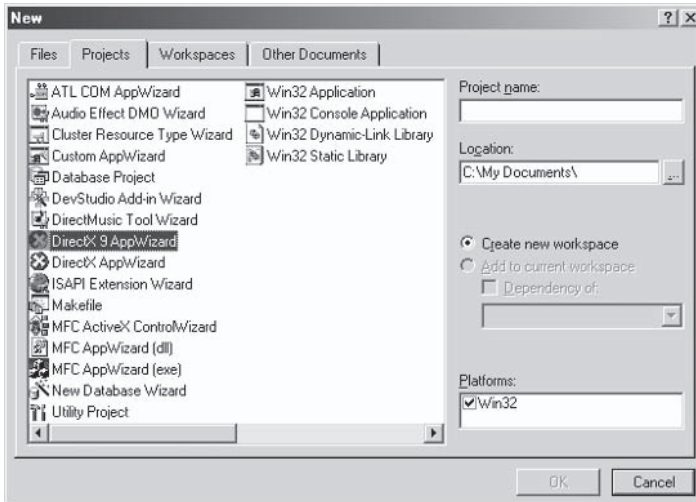


Figure 3.3



## NOTE

For your convenience I have coded a skeleton project that has everything you need to start programming a Direct3D application. This can be found on the book's CD in the Chapter 3 folder. It contains all of the following code.

```
#include <d3d9.h>
#include <d3dx9.h>
```

```
INT WINAPI WinMain(HINSTANCE hInst, HINSTANCE, LPSTR, INT)
{
    // Register the window class
    WNDCLASSEX wc = {sizeof(WNDCLASSEX), CS_CLASSDC, MsgProc, 0L, 0L,
                    GetModuleHandle(NULL), NULL, NULL, NULL, NULL,
                    "D3D", NULL};
    RegisterClassEx(&wc);

    // Create the application's window
    HWND hWnd = CreateWindow("D3D", "D3D",
                            WS_OVERLAPPEDWINDOW, 100, 100, 300, 300,
```

```

        GetDesktopWindow(), NULL, wc.hInstance, NULL);

    ShowWindow(hWnd, SW_SHOWDEFAULT);
    UpdateWindow(hWnd);

    // Enter the message loop
    MSG msg;
    while(msg.message != WM_QUIT)
    {
        if (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
        {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
        else
            Render(); //Call stuff here
    }

    UnregisterClass("D3D Tutorial", wc.hInstance);
    return 0;
}

//-----
LRESULT WINAPI MsgProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
    switch(msg)
    {
        case WM_DESTROY:
            PostQuitMessage(0);
            return 0;
        case WM_PAINT:
            return;
    }
    return DefWindowProc(hWnd, msg, wParam, lParam);
}

```

## 3.3 Programming Direct3D Applications

---

Once a window has been created and a message pump is running, Direct3D applications typically perform a number of different steps to draw images to the screen. These are listed below in the order of occurrence and are explained throughout this chapter.

### 1. Initialize Direct3D

All Direct3D applications begin with the creation of the IDirect3D9 interface and end with its deletion. It is the ultimate ancestor; the lifeline of Direct3D. A number of other interfaces descend from here. See Section 3.4.

### 2. Create a 3D device

A Direct3D device is encapsulated into an IDirect3DDevice9 interface. It represents a single graphics adapter attached to the user's computer, and really is the heart and soul of a Direct3D program. It exposes a vast assemblage of methods to determine graphics card capabilities, manipulate a 3D world, and render data to the screen, among other things. See Section 3.5.

### 3. Configure the device for rendering

As an application prepares to render each object, it must configure the Direct3D device accordingly. This is because different objects are likely to have different requirements on how they should be presented. Some might require lighting while others might require alpha blending. User interfaces will never require the former but will often need the latter. See Section 3.6.

### 4. Initialize world data

This step differs between applications, depending on their needs. It is about preparing a scene for presentation. For some it might involve loading 3D models and assigning them a position in 3D space, and for others it might involve calculating light effects. For user interfaces it will involve the initialization of all controls, such as windows, text boxes, and buttons. See Section 3.7.

## 5. Render

Once a 3D world has been initialized in the above step, an application can then begin the process of rendering it to the screen, like taking a photograph. This process occurs several times a second and is usually handled inside some procedure that a developer specifically reserves for the task. This procedure is generally referred to as the Render procedure. See Section 3.7.6.

## 6. Release Direct3D objects

All Direct3D objects, including `IDirect3D9` and `IDirect3DDevice9`, must be released on application end. This can be done quite simply by using the `Release` method of the `IUnknown` interface.

# 3.4 Initializing Direct3D

---

All Direct3D applications, from computer games to other products, must begin by instantiating an `IDirect3D9` interface and must end with its release. It represents the lifetime of a Direct3D project, and is the highest ancestor from whom a number of interfaces descend, specifically Direct3D devices as we shall see. We typically receive a pointer to a valid instance of `IDirect3D9` by calling the `Direct3DCreate9` function. Its declaration and parameter are as follows.

```
IDirect3D9* Direct3DCreate9(  
    UINT SDKVersion  
);
```

---

**UINT *SDKVersion*:** This value should always be set to `D3D_SDK_VERSION` to ensure you're working with the latest version of Direct3D.

---

Usually, `IDirect3D9` will be declared as a global pointer and will be initialized as an application begins. Some might be tempted to do this in `WinMain`, but I recommend designating a separate procedure for this task and invoking that from `WinMain`. Such modularization makes your code clearer and simpler to debug. I call such a function `InitD3D` in my projects, and this is how I refer to it throughout this book. It can be written to include a call to `Direct3DCreate9` as follows.

```
g_pD3D = Direct3DCreate9(D3D_SDK_VERSION))
    if(g_pD3D == NULL)
        //Error occurred
```

As an `IDirect3D9` object is initialized on application start, we should release this object on application end. Like before, I code a separate function to handle this. I call it `CleanUp` and it is invoked as our program receives a quit message. It can be coded to delete `IDirect3D9` as follows.

```
if(g_pD3D != NULL)
    g_pD3D->Release();
```

## 3.5 Creating a Direct3D Device — A Graphics Card

---

Direct3D devices are encapsulated into the `IDirect3DDevice9` interface and are instantiated to represent the computer's graphics card. Normally you work with just one instance of `IDirect3DDevice9` because most computers have only one graphics adapter, but of course there could be more. In terms of power and functionality, it is singularly the most important interface of Direct3D. It exposes a vast collection of methods, including functions to determine the capabilities of the graphics device, manipulate a 3D world, and render data on the screen. To create valid instances of `IDirect3DDevice9` we must call the `CreateDevice` method of `IDirect3D9`. Its declaration and parameters follow.

```

HRESULT CreateDevice
(
    UINT Adapter,
    D3DDEVTYPE DeviceType,
    HWND hFocusWindow,
    DWORD BehaviorFlags,
    D3DPRESENT_PARAMETERS* pPresentationParameters,
    IDirect3DDevice9** ppReturnedDeviceInterface
);

```

---

**UINT Adapter:** Numerical value describing which graphics card to represent. Use `D3DADAPTER_DEFAULT` to select the default hardware.

---

**D3DDEVTYPE DeviceType:** This can be one of the following values:

**D3DDEVTYPE\_HAL:** This is the recommended value. Whenever this is specified, the device will utilize hardware acceleration where possible.

**D3DDEVTYPE\_REF:** This value indicates that the device will not utilize hardware acceleration and will instead use software in most instances.

---

**HWND hFocusWindow:** Window handle to be associated with the Direct3D Device. Pass `hWnd`.

---

**DWORD BehaviorFlags:** Specifies additional information on how the device is to behave. This book uses `D3DCREATE_SOFTWARE_VERTEXPROCESSING`.

---

**D3DPRESENT\_PARAMETERS\* pPresentationParameters:** Address of a `D3DPRESENT_PARAMETERS` structure. This structure encodes information about how a device renders data. For example, it specifies whether it will be rendering data for a full-screen application or a windowed application, determines the refresh rate, and stores the window handle where data is to be rendered, along with other information. The example code for this function demonstrates how to use this structure. See the SDK for more information.

---

**IDirect3DDevice9\*\* ppReturnedDeviceInterface:** Address to return a valid `IDirect3DDevice9` interface.

---

Like IDirect3D9, we can declare IDirect3DDevice9 as a global pointer at the top of our source (`g_pd3dDevice`). It will be instantiated during the `InitD3D` function, or some other initialization procedure and released inside `CleanUp`, or somewhere else at application end. An example of initializing a Direct3D device using the `CreateDevice` method can be seen below. It is written as a continuation of procedure `InitD3D`.

```
//Structure to define how our 3D device is to operate
D3DPRESENT_PARAMETERS d3dpp;
ZeroMemory(&d3dpp, sizeof(d3dpp));
d3dpp.Windowed = TRUE;
d3dpp.SwapEffect = D3DSWAPEFFECT_DISCARD;
d3dpp.BackBufferCount = 1;

if(FAILED(g_pd3D->CreateDevice(D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL, hWnd,
                             D3DCREATE_SOFTWARE_VERTEXPROCESSING,
                             &d3dpp, &g_pd3dDevice)))
{
    return E_FAIL;
}
```

 **TIP**

To retrieve the capabilities of a Direct3D device you can call the `GetDeviceCaps` method of `IDirect3DDevice9`. You typically provide this method with a `D3DCAPS9` structure. Subsequently, this is populated with information that can be interrogated to determine what your device does and does not support. Alternatively, you can retrieve the same information from `IDirect3D9::GetDeviceCaps`.

## 3.6 Preparing to Render

---

Before an application can create, manipulate, and render objects in a 3D environment, we should structure our source code in preparation for rendering. As mentioned, rendering is the process of taking a snapshot of a 3D world from a particular vantage point at a certain point in time. To handle this process, we shall code a separate procedure exclusively for this task. This procedure will be known as `Render` and, since rendering occurs at an extremely frequent rate, it should be invoked very often from the application's message loop. Traditionally, the Windows message loop — as presented in the code from Section 3.2 — waits around for messages to occur and then acts accordingly. This is acceptable for standard applications of course, but is not best suited to the breakneck speed of games or certain multimedia products, particularly those that use Direct3D. Consequently, we'll need to amend the standard message loop such that it calls `Render` on a speedier basis rather than just when messages occur. The new, replacement message loop can be amended to call this function, as seen in the following code snippet.

```
MSG msg;
while(uMsg.message != WM_QUIT)
{
    if(PeekMessage(&uMsg, NULL, 0, 0, PM_REMOVE))
    {
        TranslateMessage(&uMsg);
        DispatchMessage(&uMsg);
    }
    else
        render(); // if there are no messages to process then Render
}
```

As `Render` is called, we begin the rendering process by clearing the screen. To do this, we invoke the `Clear` method of `IDirect3DDevice9`. Its declaration and parameters are shown here.

```

HRESULT Clear
(
    DWORD Count,
    const D3DRECT *pRects,
    DWORD Flags,
    D3DCOLOR Color,
    float Z,
    DWORD Stencil
);
    
```

---

**DWORD Count:** The number of rectangles specified in `pRects`. To clear the whole screen you should pass 0.

---

**const D3DRECT \*pRects:** List of rectangles defining screen areas to clear. Pass NULL to clear the whole screen.

---

**DWORD Flags:** Combination of one or more of the following: `D3DCLEAR_STENCIL`, `D3DCLEAR_TARGET`, and `D3DCLEAR_ZBUFFER`. These values indicate exactly what is being cleared. For example, DirectX stores specific information about how objects closer to the camera occlude those behind them. This information is stored in what is known as a depth buffer. For our purposes, however, we just pass `D3DCLEAR_TARGET` to clear the whole screen and nothing else.

---

**D3DCOLOR Color:** This argument specifies the color to which the screen should be cleared. Pass `D3DCOLOR_XRGB(0,0,255)` or some other value. See the tip later in this section.

---

**float Z:** This value has little effect on the tasks presented in this book. Pass 1.0. For those who are interested, please see the SDK.

---

**DWORD Stencil:** Again, this value is not discussed in this book. Pass 0.

---

 **TIP**

To choose colors in Direct3D we can use the `D3DCOLOR_XRGB` macro. Its arguments take the form of Red, Green, and Blue, and their values range from 0 to 255. These indicate how strongly each color is weighted, and this mix results in a final color. For example, Red is `D3DCOLOR_XRGB(255,0,0)`, Blue is `D3DCOLOR_XRGB(0,255,0)`, and Green is `D3DCOLOR_XRGB(0,0,255)`.

After the screen has been cleared, we mark the beginning and end of the rendering process by calling the `BeginScene` and `EndScene` methods. These are methods of `IDirect3DDevice9` and require no arguments. Between them a developer should render each object in the 3D world. The details of this process are explained later. Once `EndScene` is called, a developer should then call the `Present` method to finally show the scene on screen. The entire `Render` procedure therefore can be structured to render data as follows.

```
VOID Render()
{
    if(NULL == g_pd3dDevice)
        return;

    //Clear the screen
    g_pd3dDevice->Clear(0, NULL, D3DCLEAR_TARGET, D3DCOLOR_XRGB(0,0,255),
        1.0f, 0);

    // Begin the scene
    if(g_pd3dDevice->BeginScene() == D3D_OK)
    {
        // Render objects here
        g_pd3dDevice->EndScene();

        // Present the back buffer contents to the display
        g_pd3dDevice->Present(NULL, NULL, NULL, NULL);
    }
}
```

## 3.7 Initializing World Data

---

Once an application has created an `IDirect3D9` interface, and from there an `IDirect3DDevice9` object, and has configured the application as above, we can begin to initialize and position data inside a 3D world in preparation for rendering. To do this, we must represent our world data in a form that Direct3D can understand. For some applications this may mean loading a 3D model, like a

spaceship or a monster, and representing this using one of Direct3D's interfaces, in this case ID3DXMesh. For user interfaces, however, we're primarily concerned with 2D images and representing such data, like windows and controls, in terms of flat surfaces that are aligned to the screen. For these scenarios we can use one of two interfaces to express such information: Direct3D surfaces (IDirect3DSurface9) or Direct3D textures (IDirect3DTexture9). Usages of these interfaces are explained in the following subsections.

### 3.7.1 Direct3D Surfaces — IDirect3DSurface9

Surfaces are exactly what their name implies: a flat plane or a rectangle of bytes in memory. Upon this, we can hold images from files on disk like JPEGs and bitmaps, or we can even transfer data between multiple surfaces. Surfaces are encapsulated by the IDirect3DSurface9 interface, and we create them using the CreateOffscreenPlainSurface method of our Direct3D device. Remember, surfaces are blank to begin with and are not immediately drawn on screen. Instead, images are loaded onto them and are then drawn at a later point. The declaration and parameters for CreateOffscreenPlainSurface are as follows:

```
HRESULT CreateOffscreenPlainSurface
(
    UINT Width,
    UINT Height,
    D3DFORMAT Format,
    DWORD Pool,
    IDirect3DSurface9** ppSurface,
    HANDLE* pHandle
);
```

---

**UINT Width:** Surface width; this is not necessarily the width of the image you want to hold but rather the total width of the surface. It can be larger than loaded images but not smaller. The same applies for height.

---

---

**UINT Height:** Surface height.

---

**D3DFORMAT Format:** This is a D3DFORMAT constant that indicates the format of the surface to be created. Values include D3DFMT\_R8G8B8 for 24-bit images with 8 bits per channel and D3DFMT\_X1R5G5B5 for 16-bit images with 5 bits for each color and no alpha channel. Typically this value will be either D3DFMT\_A8R8G8B8 or D3DFMT\_X8R8G8B8, depending on whether the image has an alpha channel. See Section 3.8 for information on alpha channels.

---

**DWORD Pool:** Defines which resource is to hold the surface in memory. This could be in the system RAM (D3DPOOL\_SYSTEMMEM) or inside the graphics card's memory (D3DPOOL\_MANAGED). Often you will want this argument automatically selected, in which case you should pass D3DPOOL\_DEFAULT.

---

**IDirect3DSurface9\*\* ppSurface:** Address where a valid instance of IDirect3DSurface9 should be returned.

---

**HANDLE\* pHandle:** Simply pass NULL.

---

 **TIP**

Surfaces (and textures, explained later in this chapter) should be created so their dimensions are of a power of 2, such as 2x2, 4x4, 8x8, 16x16... 512x512. If they are not, it may result in anomalous stretching effects.

## 3.7.2 Direct3D Surfaces — Loading Image Files

Once a surface has successfully been created, we can then load images onto it. These images can come from many places like files on disk, system memory, and even other surfaces. This book concentrates on loading images from files. We can achieve this by calling the D3DXLoadSurfaceFromFile function. Its declaration and parameters follow.

```

HRESULT D3DXLoadSurfaceFromFile
(
    LPDIRECT3DSURFACE9 pDestSurface,
    CONST PALETTEENTRY* pDestPalette,
    CONST RECT* pDestRect,
    LPCTSTR pSrcFile,
    CONST RECT* pSrcRect,
    DWORD Filter,
    D3DCOLOR ColorKey,
    D3DXIMAGE_INFO* pSrcInfo
);
    
```

---

**LPDIRECT3DSURFACE9 pDestSurface:** Pointer to a valid surface that is to receive the loaded image.

---

**CONST PALETTEENTRY\* pDestPalette:** Pointer to a palette. This is most often NULL.

---

**CONST RECT\* pDestRect:** Using this parameter you can define a rectangle that represents a stencil into which the image will be copied. Or you can pass NULL to select the entire surface as a valid destination.

---

**LPCTSTR pSrcFile:** Valid file path specifying the image to be loaded. Acceptable file formats are .bmp, .dds, .dib, .jpg, .png, and .tga.

---

**CONST RECT\* pSrcRect:** This is a rectangle that defines a frame of pixels on the source image to be copied. Pass NULL to select the whole image.

---

**DWORD Filter:** Select how the image is filtered. For our purposes, we'll just need to pass D3DX\_FILTER\_NONE.

---

**D3DCOLOR ColorKey:** This essentially allows us to apply some transparent effects, but to achieve that we will instead use alpha blending later in the chapter. For this value just pass the default: 0xFF000000.

---

**D3DXIMAGE\_INFO\* pSrcInfo:** Pass NULL. Or, if you want to keep a copy of the image's information, pass the address of a valid D3DXIMAGE\_INFO structure.

---

**TIP**

To create a surface that matches the size and properties of an image file on disk, first call `D3DXGetImageInfoFromFile` to populate a `D3DXIMAGE_INFO` structure, and then pass this data to `CreateOffscreenPlainSurface`. See the code example below.

In this example, surfaces are declared as global variables at the tops of our sources and are initialized in procedure `InitD3D` (or whenever initialization occurs). The code below creates a surface and loads upon it an image from a file using the functions mentioned thus far.

```
D3DXIMAGE_INFO Info;
D3DXGetImageInfoFromFile("c:\image.jpg", &Info);

g_pd3dDevice->CreateOffscreenPlainSurface(Info.Width, Info.Height,
    Info.Format, &g_Surface, NULL);

D3DXLoadSurfaceFromFile(g_Surface, NULL, NULL, "c:\image.jpg", NULL,
    D3DX_FILTER_NONE, 0xFF000000, NULL);
```

### 3.7.3 Direct3D Surfaces — Rendering

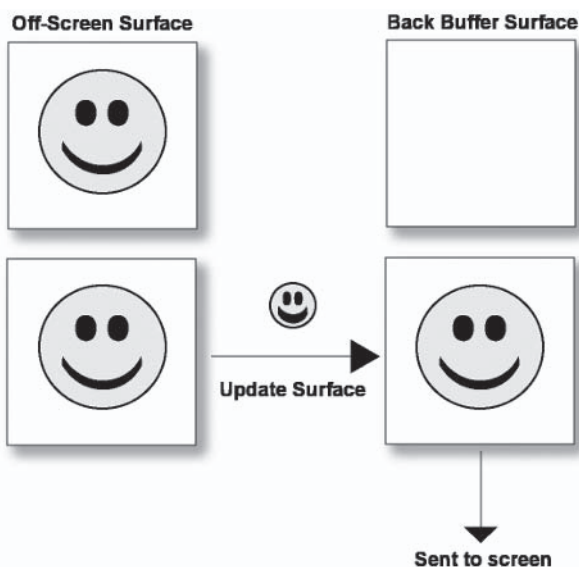


Figure 3.4

Surfaces are drawn between the `BeginScene` and `EndScene` statements of the `Render` procedure, as mentioned above. However, surfaces — like everything else in Direct3D — are not immediately drawn to the screen. Instead, Direct3D initially renders objects to a special off-screen surface, which is called the back buffer. Here, renderings continue to accumulate one by one until the `Present` method is called to complete a scene, whereupon the back buffer is pushed to the front and becomes visible on screen. Surfaces can be drawn to the back buffer by the process of surface copying — in other words, physically copying the pixels from a source surface and pasting them onto a destination surface, the back buffer. Surface copying can be achieved with the `UpdateSurface` method of `IDirect3DDevice9`. Its declaration and parameters are as follows.

```
HRESULT UpdateSurface
(
    IDirect3DSurface9* pSourceSurface,
    CONST RECT* pSourceRect,
    IDirect3DSurface9* pDestinationSurface,
    CONST POINT* pDestinationPoint
);
```

---

**IDirect3DSurface9\* pSourceSurface:** Pointer to the source surface where pixels are copied from. This will be your standard image surface.

---

**CONST RECT\* pSourceRect:** Defines a rectangular frame that selects a subset of pixels to copy. Pass `NULL` to select the whole of the surface.

---

**IDirect3DSurface9\* pDestinationSurface:** Pointer to a destination surface that is to receive the copied pixels. This will often be a pointer to the back buffer, although it could be other surfaces. For information on how to retrieve a pointer to the back buffer, see below.

---

**CONST POINT\* pDestinationPoint:** A 2D point whose origin is the top left. It defines the destination where image pasting should begin.

---

The `UpdateSurface` method requires, as its third argument, a pointer to the destination, which is the back buffer. To retrieve this pointer we must call the `GetBackBuffer` method of `IDirect3DDevice9`. See below for its function declaration and parameters.

```
HRESULT GetBackBuffer
(
    UINT iSwapChain,
    UINT BackBuffer,
    D3DBACKBUFFER_TYPE Type,
    IDirect3DSurface9 **ppBackBuffer
);
```

---

**UINT *iSwapChain*:** Pass 0.

---

**UINT *BackBuffer*:** For our purposes, just pass 0. For more information, please consult the SDK.

---

**D3DBACKBUFFER\_TYPE *Type*:** Pass `D3DBACKBUFFER_TYPE_MONO`.

---

**IDirect3DSurface9 \*\**ppBackBuffer*:** The address where a valid back buffer pointer is to be returned. Don't forget to release it when you're finished.

---

The code below draws a surface to screen via the back buffer. You'll notice that the back buffer itself has been declared as a local variable of the `Render` procedure.

```
VOID Render()
{
    LPDIRECT3DSURFACE9 BackBuffer = NULL;
    // Clear the back buffer to a blue color
    g_pd3dDevice->Clear(0, NULL, D3DCLEAR_TARGET, D3DCOLOR_XRGB(0,0,255),
        1.0f, 0);
    // Begin the scene
    if(SUCCEEDED(g_pd3dDevice->BeginScene()))
    {
        g_pd3dDevice->GetBackBuffer(0,0,D3DBACKBUFFER_TYPE_MONO, &BackBuffer);
        g_pd3dDevice->UpdateSurface(g_Surface, NULL, BackBuffer, NULL);
    }
}
```

```

if(BackBuffer != NULL)
    BackBuffer->Release();
// End the scene
g_pd3dDevice->EndScene();
}
// Present the back buffer contents to the display
g_pd3dDevice->Present(NULL, NULL, NULL, NULL);
}

```

### 3.7.4 Direct3D Textures — IDirect3DTexture9

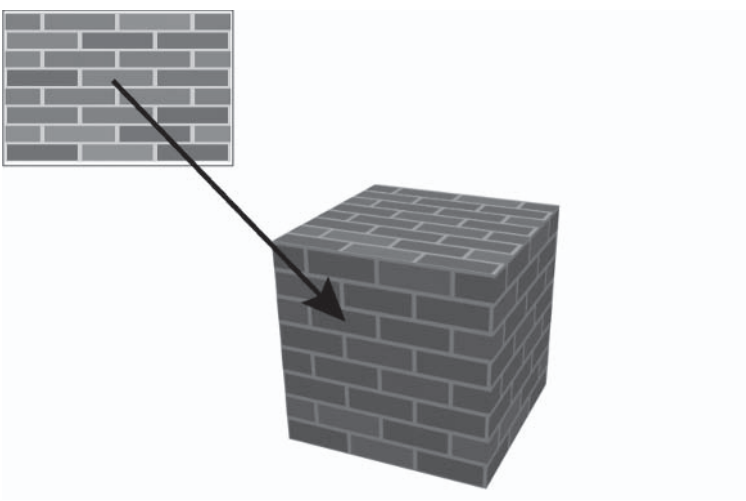


Figure 3.5

In addition to surfaces, Direct3D provides textures to present flat images on screen. Textures are really an advanced surface that can perform a number of effects. They are encapsulated into the `IDirect3DTexture9` interface and can be moved, scaled, and rotated about the screen, as well as mapped onto the skin of 3D models. We can create and load them from files all at once by calling the `D3DXCreateTextureFromFile` function. Its declaration and parameters are listed below.

```
HRESULT D3DXCreateTextureFromFile  
(  
    LPDIRECT3DDEVICE9 pDevice,  
    LPCTSTR pSrcFile,  
    LPDIRECT3DTEXTURE9 *ppTexture  
);
```

---

**LPDIRECT3DDEVICE9 pDevice:** Pointer to the Direct3D device. Usually something like `g_pd3dDevice`.

---

**LPCTSTR pSrcFile:** Path to an image file.

---

**LPDIRECT3DTEXTURE9 \*ppTexture:** Address where a valid texture interface is returned.

---

### 3.7.5 Direct3D Textures — Preparing to Render

Before we can draw textures to the screen we must understand that they're distinctive from surfaces in what they achieve and how they operate. As a result, we can't copy them directly to the back buffer like we can with surfaces. Instead, we must use a separate interface called `ID3DXSprite`. `ID3DXSprite` acts like a GDI pen — or any normal pen for that matter — and can draw any number of textures straight to the back buffer for us. Instances of `ID3DXSprite` are created by calling the `D3DXCreateSprite` function. Its declaration and parameters appear below.

```
HRESULT D3DXCreateSprite  
(  
    LPDIRECT3DDEVICE9 pDevice,  
    LPD3DXSPRITE *ppSprite  
);
```

---

**LPDIRECT3DDEVICE9 pDevice:** Pointer to the Direct3D device.

---

**LPD3DXSPRITE \*ppSprite:** Address where a valid `ID3DXSprite` interface will be returned.

---

**TIP**

Remember, you only need one instance of ID3DXSprite, no matter how many textures you have.

Typically, textures and the ID3DXSprite interface would be declared as global variables at the top of our source files, and both would be created during application initialization. The code to create a texture from a file and initialize an ID3DXSprite object can be seen in the following code extract. It is coded inside the InitD3D initialization function.

```
//Create a texture
D3DXCreateTextureFromFile(g_pd3dDevice, "c:\\image.bmp", &g_Texture);

//Create a sprite
D3DXCreateSprite(g_pd3dDevice, &g_Sprite);
```

### 3.7.6 Direct3D Textures — Rendering

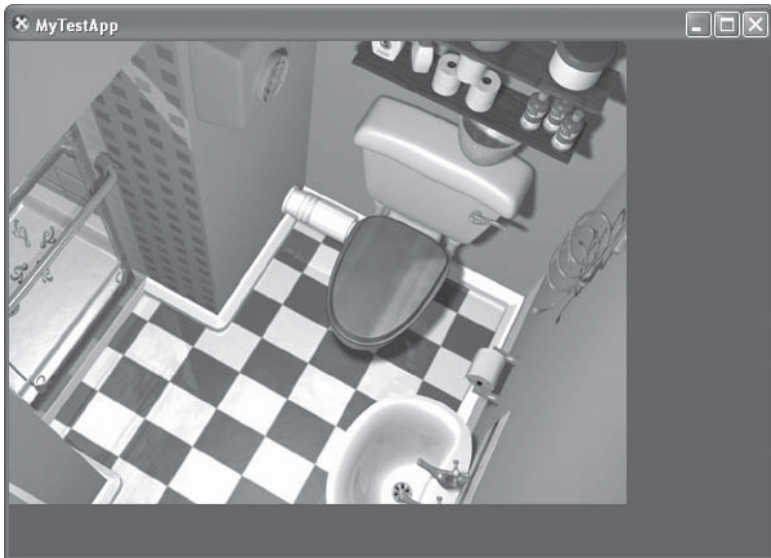


Figure 3.6

Like surfaces, textures are drawn to the back buffer. For textures, this is done by `ID3DXSprite`, between the `BeginScene` and `EndScene` statements of the rendering loop. To do this we must call upon a series of functions. First, we must call the `Begin` method of `ID3DXSprite` to prepare the device for rendering textures. Its function declaration and parameter follow.

```
HRESULT Begin(  
  
    DWORD Flags  
);
```

---

**DWORD *Flags*:** This can be 0 and/or a combination of the following values. For this example, just pass 0.

```
D3DXSPRITE_DONOTSAVESTATE  
D3DXSPRITE_DONOTMODIFY_RENDERSTATE  
D3DXSPRITE_OBJECTSPACE  
D3DXSPRITE_BILLBOARD  
D3DXSPRITE_ALPHABLEND  
D3DXSPRITE_SORT_TEXTURE  
D3DXSPRITE_SORT_DEPTH_FRONTTOBACK  
D3DXSPRITE_SORT_DEPTH_BACKTOFRONT
```

---

Secondly, after `Begin` has been called, we must set the sprite's transformation using the `SetTransform` method. This means that we must tell `ID3DXSprite` the position, orientation, and scale at which we wish to draw the texture. We can express each of these values — position, scale, and rotation — as `D3DXVECTOR2` structures, but we must use the `D3DXMatrixTransformation2D` function to compile them into a data structure that the `SetTransform` method can understand. This structure is called a matrix and is of type `D3DXMATRIX`. Thus, to tell `ID3DXSprite` about our transformations, we should first call `D3DXMatrixTransformation2D` and then pass its returned structure to `SetTransform`. The function declaration and parameters for these functions are shown below.

```

D3DXMATRIX *WINAPI D3DXMatrixTransformation2D(
    D3DXMATRIX *pOut,
    CONST D3DXVECTOR2 *pScalingCenter,
    FLOAT *pScalingRotation,
    CONST D3DXVECTOR2 *pScaling,
    CONST D3DXVECTOR2 *pRotationCenter,
    FLOAT Rotation,
    CONST D3DXVECTOR2 *pTranslation
);
    
```

---

**D3DXMATRIX \*pOut:** Address of a D3DMATRIX structure. This will represent the accumulative transformation — in other words, a combination of scaling, rotation, and translation — that will be applied to the drawn texture.

---

**CONST D3DXVECTOR2 \*pScalingCenter:** Center for scaling to occur. Just pass NULL.

---

**FLOAT \*pScalingRotation:** Scaling rotational value. Again, for our purposes, just pass NULL.

---

**CONST D3DXVECTOR2 \*pScaling:** Address of a D3DXVECTOR2 structure defining the scaling to occur on the texture's X and Y axes. To maintain the original scale, both axes should be 1. To halve the scale, use 0.5; to double the scale, use 2.

---

**CONST D3DXVECTOR2 \*pRotationCenter:** Address of a D3DXVECTOR2 structure defining the center about which rotation occurs.

---

**FLOAT Rotation:** The amount of rotation to be applied in radians.

---

**CONST D3DXVECTOR2 \*pTranslation:** Address of a D3DXVECTOR2 structure defining the displacement of the texture, in pixels, from the origin of the screen.

---

```
HRESULT SetTransform(  
    CONST D3DXMATRIX *pTransform  
);
```

---

**CONST D3DXMATRIX \*pTransform:** Address of a matrix representing a series of transformations that must be applied to the drawn texture.

---

Once the transformation has been set, you can actually start drawing a texture. This is achieved by calling the Draw method of ID3DXSprite. Remember, all textures drawn with this method will be transformed according to the transform we set in the previous step, using SetTransform. The Draw method below also accepts translation parameters, etc. However, since we already set them in the previous stage, we can pass NULL for them here. Its function declaration and parameters appear below.

```
HRESULT Draw(  
    LPDIRECT3DTEXTURE9 pSrcTexture,  
    CONST RECT *pSrcRect,  
    CONST D3DXVECTOR3 *pCenter,  
    CONST D3DXVECTOR3 *pPosition,  
    D3DCOLOR Color  
);
```

---

**LPDIRECT3DTEXTURE9 pSrcTexture:** Pointer to the texture to render.

---

**CONST RECT \*pSrcRect:** This defines a subrectangle from the texture to draw. Pass NULL to select the entire texture.

---

**CONST D3DXVECTOR2 \*pCenter:** Defines the center of the texture. Passing NULL is considered the same as specifying (0,0).

---

**CONST D3DVECTOR2 \*pPosition:** A 2D vector that defines how far the texture is moved across the screen in pixels on each of the X and Y axes. The origin is (0,0), at the top-left corner of the screen. Pass NULL if you're using a transform matrix.

---

---

**D3DCOLOR Color:** Passing 0xFFFFFFFF maintains the standard image color settings. It can be other colors, however. To do this, please see the SDK for more information.

---

Finally, the last stage in rendering a texture involves calling the End method of ID3DXSprite to complete the process. This requires no arguments. Thus, the entire Render procedure can be coded to draw a texture at a specific position and orientation on screen, as follows.

```
VOID Render()
{
    if(NULL == g_pd3dDevice)
        return;

    g_pd3dDevice->Clear(0, NULL, D3DCLEAR_TARGET, D3DCOLOR_XRGB(0,0,255),
        1.0f, 0);

    // Begin the scene
    if(SUCCEEDED(g_pd3dDevice->BeginScene()))
    {
        D3DXVECTOR2 Translation;
        Translation.x = 500;
        Translation.y = 500;
        D3DXVECTOR2 Scaling;
        Scaling.x = 1.0f;
        Scaling.y = 1.0f;

        D3DXMATRIX Mat;
        D3DXMatrixTransformation2D(&Mat, NULL, 0, &Scaling, NULL, 0, &
            Translation);

        g_Sprite->Begin(0);
        g_Sprite->SetTransform(&Mat);
        g_Sprite->Draw(g_Texture, NULL, NULL, NULL, 0xFFFFFFFF);
        g_Sprite->End();

        //End the scene
        g_pd3dDevice->EndScene();
    }
}
```

```
// Present the scene  
g_pd3dDevice->Present(NULL, NULL, NULL, NULL);  
}
```

## 3.8 Alpha Blending

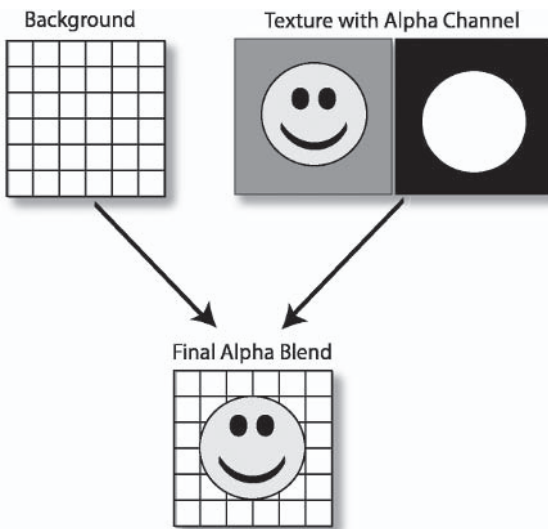


Figure 3.7

One of the useful effects that we can perform with textures is alpha blending. Basically, certain images can contain additional information known as an alpha channel. This essentially can be thought of as a mask, or a map, that is associated with the image itself and which matches its dimensions in terms of width and height. It consists of a range of grayscale colors, from white to black, and these determine the visibility of pixels in the corresponding image. Specifically, they define whether corresponding pixels are fully visible, partly transparent, or fully transparent. Therefore, Direct3D uses alpha blending to show images according to the settings of their alpha channel. This means that it is particularly useful for creating interfaces with transparent controls, misted buttons, and similar

effects. See Figure 3.7 to visualize this. Images containing alpha channels are created in many different ways. This chapter demonstrates how to do so using image editing software such as Adobe Photoshop, Paint Shop Pro, and the DirectX Texture Tool.

### 3.8.1 Using Adobe Photoshop

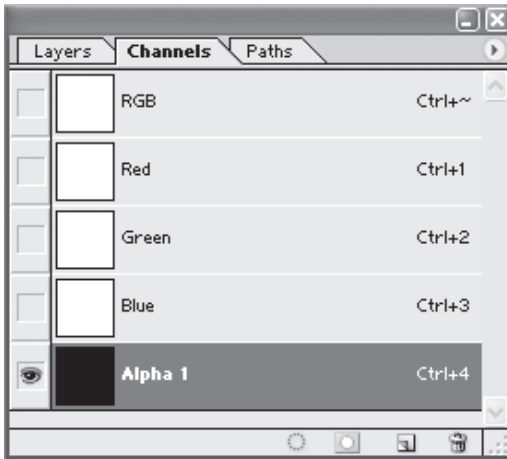


Figure 3.8

If you can afford it, Adobe Photoshop is a great way to create alpha channels. It uses the Channels window to present a visual list of valid image channels, like Red, Green, and Blue. This is also where the alpha channel will appear. To add one to a new or existing image, we must perform the following steps:

1. Create or load an image using **File | New** or **File | Open**, as usual.
2. Ensure the Channels window is visible from the Windows menu. This lists all channels present in the image and is where we add an alpha channel.
3. Click the **New Channel** button (third button from the left at the bottom of Figure 3.8).

4. Now, you can use the normal editing and painting tools to create an alpha channel. Please remember to save your image in a format that supports alpha channels, such as .tga; otherwise, the effects will not be visible in your applications.

### 3.8.2 Using Paint Shop Pro

Jasc Software's Paint Shop Pro is another excellent image editing tool. In addition, its inexpensive and trial versions are available online. There is also an evaluation version on the companion CD. To create an alpha channel in this program, you should perform the following steps:

1. Create two new images or load two images from existing files. One will be the alpha channel (mask) and the other will be the masked image. To create new images, simply click **File | New**, then fill in the dimensions and color properties. For alpha channels, the color depth should be GreyScale (8 Bit).
2. To make an image the alpha channel of the other, make sure the main image's window is active. Then click **Layers | New Mask Layer | From Image**. A dialog appears in which a drop-down box lists the currently active images that can be selected to be an alpha channel. Here, choose whichever is required.
3. Because Paint Shop Pro distinguishes masks from alpha channels, you must now save the mask to the image's alpha channel. To do this, click **Layers | Load/Save Mask | Save Mask to Alpha Channel**. Assign it a name and click **OK**.
4. Finally, we just need to save the entire image to disk. Click **File | Save**, and select a file format that supports alpha channels, such as .tga. Click the **Options** button and ensure the settings are 24 Bits, Uncompressed. Click **OK** and then **Save**. Voilà!

### 3.8.3 Using the DirectX Texture Tool

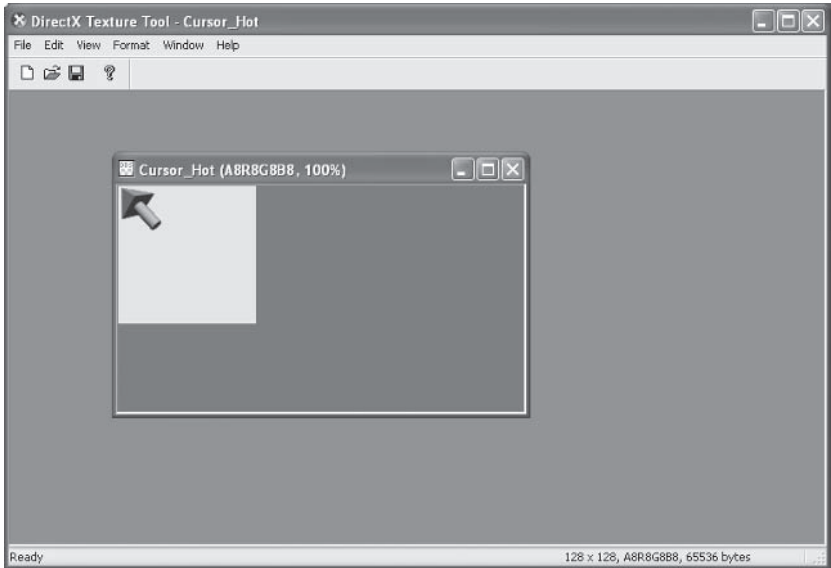


Figure 3.9

The DirectX Texture Tool does not append an alpha channel to an image. Rather, it takes two separate images, one for the alpha channel and one for the image itself, and collates them together as a new image file. To do this we must perform the following steps:

1. Create two separate images, one for the alpha channel and the other for the image. Place the files in the same folder and name them identically, but add “\_a” to the channel name, such as Image.bmp and Image\_a.bmp.
2. Load the DirectX Texture Tool and open either one of the images.
3. To converge them in one file, just click **Save As** and save to a new file.

### 3.8.4 Enabling Alpha Blending in Direct3D

Direct3D does not implicitly enable alpha blending, which means that such effects may not immediately be visible in our applications. We do not need to adjust the texture or ID3DXSprite interfaces themselves to enable this, but we do need to set a flag about how our Direct3D device renders images. To enable alpha blending, we use the SetRenderState function of IDirect3DDevice9. This can be used to set a whole assemblage of rendering behaviors, not just alpha blending. Its declaration and parameters follow.

```
HRESULT SetRenderState  
(  
    D3DRENDERSTATETYPE State,  
    DWORD Value  
);
```

---

**D3DRENDERSTATETYPE *State*:** This can be one of many constants, each of which represents a specific setting. A whole list of these can be found in the DirectX SDK at DirectX Graphics\Reference\Enumerated Types\D3DRENDERSTATETYPE. To enable alpha blending, this value should be D3DRS\_ALPHABLENDENABLE.

---

**DWORD *Value*:** This is the actual value that *State* should be set to. For alpha blending, this value should be either TRUE or FALSE to indicate whether it should be enabled or disabled, respectively. Pass TRUE.

---

Once alpha blending has been set to TRUE, you must also set two further states using SetRenderState: D3DRENDERSTATE\_SRCBLEND and D3DRENDERSTATE\_DESTBLEND. These define how the alpha blending process is to be performed. The following InitD3D function has been coded to include a call to SetRenderState. It sets alpha blending to TRUE and configures the other states as required. From this point onward, alpha blending effects will be visible in all of our renderings.

```
g_pd3dDevice->SetRenderState(D3DRS_ALPHABLENDENABLE, true);  
g_pd3dDevice->SetRenderState(D3DRENDERSTATE_SRCBLEND, D3DBLEND_SRCCOLOR);  
g_pd3dDevice->SetRenderState(D3DRENDERSTATE_DESTBLEND,  
                             D3DBLEND_INVSRCCOLOR);
```

 **TIP**

When alpha blending is not needed, it is advisable to disable the setting by passing FALSE to SetRenderState. This is because alpha blending causes a reduction in rendering speed when it is activated.

## 3.9 Conclusion

Because the technical scope of this chapter is so vast, and because DirectX is not so much a difficult API to use as it is to learn, some readers will probably need to reread through specific areas to perhaps clarify one or two points I've explained. Ultimately, it is important to have understood with relative confidence the information I've included in this chapter before moving on because it provides the working foundations for all the graphical tasks and chores presented from here on. Not understanding these basics could lead to confusion later. The best advice I can give to those looking for additional information is to read through the DirectX SDK documentation and complete the tutorial projects, and search Internet forums and sites like GameDev and FlipCode for relevant articles or tutorials on DirectX graphics and Direct3D. Overall, in presenting an introduction to DirectX graphics we have now garnered sufficient knowledge to draw a complete user interface, but this alone is not enough. We must also respond to user input via the keyboard and mouse, and using Direct Input to achieve this is the topic we explore in the next chapter.

This page intentionally left blank.

# Introducing DirectInput

This chapter is about controlling user input devices, like keyboards and mice, and about reading information from them, such as which buttons the user pressed or the position of the mouse cursor. To achieve this we use the DirectX API known as DirectInput. Specifically, we will learn how to do the following in this chapter:

- Create COM interfaces that represent and control input peripherals attached to the user's computer.
- Set how input devices operate and are shared from program to program.
- Read information from keyboards and mice whenever events occur. This will consist of determining which keyboard keys or mouse buttons are pressed, and reading the position of the mouse cursor.
- Set the mouse cursor to an image loaded from a file.

## 4.1 DirectInput Basics

---

Like Direct3D, DirectInput is a collection of COM interfaces. This collection allows an application to intuitively read information about the current status of a wide number of input peripherals, even when the application is running in the background. These peripherals include the mouse, keyboard, joystick, and other game controllers such as force feedback devices. Figure 4.1 demonstrates a simplified structure of DirectInput.

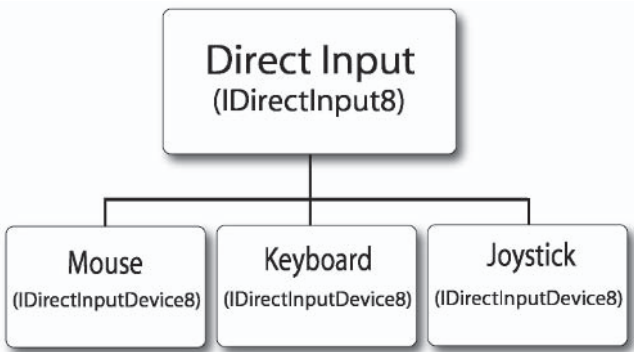


Figure 4.1

## 4.2 Getting Started

---

Starting a DirectInput application in Microsoft Visual C++ is simple. Just click File | New and follow the standard Windows application techniques: Create a window, create a message loop, etc. Or, if you took my advice in the previous chapter and created a template project, you can load it up from there. One thing we do need to be sure about is including the appropriate header (`dinput.h`) and the corresponding library file (`dinput.lib`). The steps to create a project can be seen in Figures 4.2 and 4.3.

```

// file: interface.cpp
//
// Desc: DirectX window application created by the DirectX AppWizard
//-----
#define STRICT
#include <windows.h>
#include <comactrl.h>
#include <comdlg.h>
#include <basetsd.h>
#include <math.h>
#include <stdio.h>
#include <d3dx9.h>
#include <dxerr9.h>
#include <tchar.h>
#include "DXUtil.h"
#include "D3DEnumeration.h"
#include "D3DSettings.h"
#include "D3DApp.h"
#include "D3DFont.h"
#include "D3DUtil.h"
#include "resource.h"

#include <input.h>

//-----
// Global access to the app (needed for the global WndProc())
//-----
CMyD3DApplication* g_pApp = NULL;
HINSTANCE g_hInst = NULL;

CCTest* a = NULL;
CCTest2* b = NULL;
CCTest2* c = NULL;

```

Figure 4.2

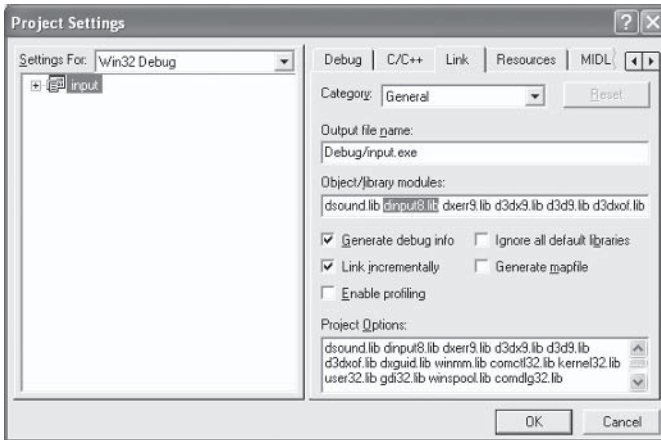


Figure 4.3

**NOTE**

In this book we are just programming the keyboard and mouse devices, and in such circumstances DirectInput provides us with no distinctive advantages over the standard Win32 functions. It is only when using joysticks and all manner of input devices that DirectInput really shines. This book is about DirectX and user interfaces, and so it is for this reason I have chosen to handle input using DirectInput. However, for those who prefer the standard Win32 functions, there is no reason why they can't be used instead.

**TIP**

Please remember that applications wishing to use IME or Unicode character sets cannot use DirectInput because it doesn't support localization.

## 4.3 Programming

---

Once an application is configured to use DirectInput we can start programming our input devices. To do this, an application typically performs several steps. These steps are explained and listed in the order in which they are performed. They are explained in more detail as the chapter progresses.

### 1. Create a DirectInput object

All DirectInput applications begin by initializing the topmost COM interface, `IDirectInput8`, and end with its release. From here, all other interfaces are created. See Section 4.4.

### 2. Create a DirectInput device object(s)

A DirectInput device object (`IDirectInputDevice8`) is a COM interface representing a single device, like a keyboard. One interface per device is needed, and the properties of a specific device are read and manipulated through the methods of this interface. See Section 4.5.

### 3. Configure the DirectInput device object(s)

After creating an instance of `IDirectInput8` in step 1, and from that an instance to a device through `IDirectInputDevice8` in step 2, the device in question must be initialized with a set of starting properties. These vary between devices. They include setting the cooperative level, whose values like `Exclusive` and `Non-Exclusive` determine how the device is shared between applications, and setting the data format, whose properties set the type of data we're dealing with, such as keyboard buttons, mouse buttons, and joystick axes. See Section 4.6.2.

#### 4. Acquire the device

Then, an application notifies `DirectInput` that it is ready to receive and process input data from a device by calling the `IDirectInputDevice8::Acquire` method. Upon success, the device is ready to be polled. See Section 4.6.3.

#### 5. Read data from the device

Polling is the process of reading information from a device. This can be thought of as taking a series of snapshots — like freeze frames of its current state — and then polling that state with a number of questions, such as “Is mouse button 1 pressed?” The snapshot process occurs at regular intervals, such as the `Render` procedure, or even another that we could add. This will ensure that an updated image of the device is maintained. See Sections 4.6.3 and 4.7.3.

#### 6. Release the `DirectInput` and `DirectInput` device object(s)

All COM interfaces of `DirectInput`, like `IDirectInput8` and `IDirectInputDevice8`, must be freed from memory before the application closes, using the normal `IUnknown::Release` method. See Section 4.4.

## 4.4 Creating a `DirectInput` Object

---

All `DirectInput` applications, regardless of the devices we’re attempting to program, must begin with the instantiation of the `IDirectInput8` COM interface, and end with its deletion. This is the topmost interface that marks the duration of `DirectInput`’s lifetime and is the parent from which all other `DirectInput` interfaces descend. We receive a valid pointer to an instance of `IDirectInput8` using the `DirectInputCreate8` function. Its declaration and parameters are as follows:

```

HRESULT WINAPI DirectInput8Create
(
    HINSTANCE hInst,
    DWORD dwVersion,
    REFIID riidIif,
    LPVOID *ppvOut,
    LPUNKNOWN punkOuter
);

```

---

**HINSTANCE *hInst***: Instance handle to the application or DLL that is creating the DirectInput interface. In most cases this will be the `g_hInst` global variable declared at the top of the source.

---

**DWORD *dwVersion***: A dword value indicating the version of DirectInput you want to work with. Specify the constant `DIRECTINPUT_VERSION` for the latest.

---

**REFIID *riidIif***: A unique identifier defining whether to retrieve the Unicode or ANSI version of IDirectInput. Specifying `IID_IDirectInput8` will auto-select the appropriate one.

---

**LPVOID *\*ppvOut***: The address where a valid IDirectInput8 interface is to be returned. In our case this will be an `LPDIRECTINPUT8` pointer, and it will be declared as a public member of `CMyD3DApplication`.

---

**LPUNKNOWN *punkOuter***: Reserved; pass `NULL`.

---

The best place to declare an instance of IDirectInput8 is as a global pointer at the top of our source, along with all of the other objects. I name this `g_lpDI`. One place we could initialize it is inside `InitD3D`, but the most ideal solution would be to code a separate procedure exclusively for initializing DirectInput interfaces. This will be called `InitInput`, and will be invoked just before the `InitD3D` function during the message loop. It can be written to create a DirectInput object as follows.

```

DirectInput8Create(g_hInst, DIRECTINPUT_VERSION,
                  IID_IDirectInput8, (void**)&g_lpDI, NULL);

```

Consequently, we free this interface in `CleanUp`, which is called on application close. It can be written as follows:

```
if(g_lpDI != NULL)
    g_lpDI->Release();
```



**NOTE**

It is natural to ask why `IDirectInput8`, and later `IDirectInputDevice8`, are appended with the number 8 rather than 9 when we are supposed to be using DirectX 9. This is because there have been no changes made to these specific interfaces since the previous version.

## 4.5 Creating DirectInput Devices

After successfully creating an `IDirectInput8` COM interface we typically invoke its `CreateDevice` method to create a `DirectInput` device, represented by an `IDirectInputDevice8` interface. Each of these encapsulate a single input device, such as a keyboard, mouse, or joystick. So we must create as many of them as we need, one per device. We can read information from them — such as which mouse button was pressed — and set properties about them — such as which other applications can share them — using the methods of this interface. Let’s see the declaration and parameters for `CreateDevice` to better understand how we create them.

```
HRESULT CreateDevice
(
    REFGUID rguid,
    LPDIRECTINPUTDEVICE *lpDirectInputDevice,
    LPUNKNOWN pUnkOuter
);
```

---

**REFGUID *rguid*:** A reference to a GUID representing the device to create. For us this value will be either `GUID_SysKeyboard` or `GUID_SysMouse`, indicating the default keyboard and mouse, respectively. However, other values can be returned from the `EnumDevices` call (explained in a moment) for specifying other devices, such as numerous joystick inputs.

---

---

**LPDIRECTINPUTDEVICE \*lpDirectInputDevice:** An address where a valid DirectInput device, matching the creation parameters, should be returned to upon success.

---

**LPUNKNOWN pUnkOuter:** Just pass NULL. If you really want more information, see the DirectX documentation.

---

 **TIP**

To determine which devices other than the standard mouse and keyboard might be connected to the user's computer, we call `IDirectInput8::EnumDevices`, which sequentially lists a number of devices matching a search criteria and returns their interface ID. This can then be passed onto the `CreateDevice` method as the `rguid` parameter to create an `IDirectInputDevice8` interface representing that device.

Having examined the `CreateDevice` function in more detail, we can now proceed to more of a practical implementation by creating and configuring two working devices over the next few sections. These are the keyboard and the mouse devices.

## 4.6 The Keyboard

---

The following subsections describe how to create, configure, and read from the keyboard.

### 4.6.1 Creating the Keyboard

An application prepares to program a keyboard with DirectInput by calling `CreateDevice`. The address that it fills with a valid `IDirectInputDevice8` interface can be declared as a global pointer at the top of the source. I have called it `g_Keyboard`. The declarations can be written like this:

```
LPDIRECTINPUT8 g_lpDI;           //DirectInput object
LPDIRECTINPUTDEVICE8 g_Keyboard; //DirectInput device
```

These two pointers are set to NULL, and both are initialized to valid interfaces inside `InitInput`. `g_lpDI` is initialized using `DirectInputCreate8` and `g_Keyboard` using `CreateDevice`. The entire code for this is as follows.

```
//Create the DirectInput object
DirectInput8Create(g_hInst, DIRECTINPUT_VERSION,
                  IID_IDirectInput8, (void*)&g_lpDI, NULL);

//Create the keyboard device object
g_lpDI->CreateDevice(GUID_SysKeyboard, & g_Keyboard, NULL);
```

## 4.6.2 Configuring the Keyboard

After a `DirectInput` object has been created successfully, and from there a `DirectInput` device, we must then configure our valid device — in this case a keyboard — with several properties that describe how it is to behave. Firstly, this involves setting the device's cooperative level using the `SetCooperativeLevel` method of `IDirectInputDevice8`. This method, whose parameters consist of constants such as `Exclusive` and `Non-Exclusive`, define how the keyboard device is to be shared between other applications. Its function declaration and parameters are as follows:

```
HRESULT SetCooperativeLevel
(
    HWND hwnd,
    DWORD dwFlags
);
```

---

**HWND *hwnd*:** Handle of the window which is to be associated with the device. This is often referred to as the input focus window. Commonly, you will pass `g_hWnd` as this parameter.

---

**DWORD *dwFlags*:** This can be a combination of one or more of the following constants that indicate how the device will be shared.

**DISCL\_BACKGROUND:** The device can be read from at any time, even when the application is not active.

**DISCL\_EXCLUSIVE:** No other application can obtain exclusive access to a device while another holds it. When mice and keyboards are used in this mode, a programmer must be certain to unacquire the device when the Windows messages `WM_ENTERSIZEMOVE` or `WM_ENTERMENULOOP` are received; otherwise, the devices will not operate properly. (See the DirectX SDK for more information.)

**DISCL\_FOREGROUND:** Foreground access means that a device will be unacquired (data will not be received) when the application is no longer active, such as when another window receives focus.

**DISCL\_NONEXCLUSIVE:** Access in Non-Exclusive mode has no repercussions on how other applications request to use the same device.

**DISCL\_NOWINKEY:** Disables the Windows logo key so that a user cannot break out of an application.

---

After setting the cooperative level of a device, we proceed to set its data format using the `SetDataFormat` method of `IDirectInputDevice8`. This function is the last stage in device configuration and accepts a `DIDATAFORMAT` structure, whose properties are used to describe the type of information we're expecting the device to communicate back to us when user input occurs, such as mouse button presses and cursor positions for mice or key states for keyboards. The function declaration and parameter for `IDirectInputDevice8::SetDataFormat` are as follows:

```
HRESULT SetDataFormat  
(  
    LPCDIDATAFORMAT lpdf  
);
```

---

**LPCDIDATAFORMAT *lpdf*:** The address of a `DIDATAFORMAT` structure. An application can define its own or use one of a predefined set for the most common types. These include: `c_dfDIKeyboard`, `c_dfDIMouse`, `c_dfDIMouse2`, `c_dfDIJoystick`, and `c_dfDIJoystick2`. For the purposes of this book, we will use the first two.

---

This section has demonstrated how devices are configured using `SetDataFormat` and `SetCooperativeLevel`. These can be coded after a device's creation inside `InitInput`. The entire procedure therefore looks as follows.

```

void InitInput(void)
{

//Create DirectInput object
HRESULT Result = DirectInput8Create(g_hInst, DIRECTINPUT_VERSION,
                                   IID_IDirectInput8, (void**)&g_lpDI, NULL);

if(SUCCEEDED(Result))
{

//Create DirectInput device
Result = g_lpDI->CreateDevice(GUID_SysKeyboard, &g_lpDIDevice, NULL);

if(SUCCEEDED(Result))
{
//Set cooperative level
g_lpDIDevice->SetCooperativeLevel(g_hWnd, DISCL_FOREGROUND |
DISCL_NONEXCLUSIVE);

//Set data format
g_lpDIDevice->SetDataFormat(&c_dfDIKeyboard);

}
else
{
    MessageBox(g_hWnd,"Direct Device Failed","",MB_OK);
    PostQuitMessage(0);
}
}
else
{
    MessageBox(g_hWnd,"DirectInput Failed","",MB_OK);
    PostQuitMessage(0);
}
return S_OK;
}
    
```

### 4.6.3 Reading from the Keyboard

Once a device such as the keyboard has been created and configured using the methods previously illustrated, we are then prepared to read user input from it. This occurs at regular intervals — like the rendering loop — to ensure that an updated image of the device is maintained. On each loop, reading begins by acquiring the device with the `IDirectInputDevice8::Acquire` method. This notifies the device that we're ready to receive a new stream of user input from it, and in turn the method returns — through success or failure — a value indicating whether it can currently provide us with data. It might fail when another application already has exclusive access to the device or when our own application no longer is active, depending on the creation parameters that were provided to `SetCooperativeLevel`. The function declaration for the `Acquire` method looks like this:

```
HRESULT Acquire(VOID);
```

Upon successful acquisition of a device we proceed to read its data by taking a snapshot of it, like a freeze-frame. Afterward, we can query this snapshot to determine its status by asking it a number of questions, like “Is mouse button 1 pressed?” or, more pertinently for keyboards, “Is button ‘A’ pressed?” In DirectX terms, this is known as immediate data. We use the `GetDeviceState` method of `IDirectInputDevice8` to take a snapshot of the device at the time of the call. The snapshot is then returned into a data structure, which for keyboards is a 256-byte array. The `GetDeviceState` declaration and parameters are:

```
HRESULT GetDeviceState  
(  
    DWORD cbData,  
    LPVOID lpvData  
);
```

---

**DWORD *cbData***: Size of parameter *lpvData* in bytes.

---

**LPVOID lpvData:** Here, the snapshot of the device is sent to the address of a data structure, whose format varies according to the device. For a mouse it should be `DIMOUSESTATE`, and for a keyboard it is an array of 256 bytes, representing the pressed state of each key on the keyboard.

Finally, once a snapshot of the device has been taken using `GetDeviceState`, we can proceed to question the properties of the returned structure in order to ascertain the device's state. For keyboards we can use the following `#define` macro that we can use like a function. This accepts a single key and the returned structure as arguments, and returns `TRUE` or `FALSE` to indicate whether the key was pressed in that snapshot or not. The macro looks like this, and we can paste it somewhere at the top of our source file.

```
#define KEYDOWN(name, key) (name[key] & 0x80)
```

Now that we have examined the theoretical stages in reading input from a device, and since we have seen how data must be read through snapshots on regular intervals, we can begin to implement all of the details in this section into code. Like initialization, we will add a separate procedure for reading input. This will be called just before the `Render` function during our message loop, and will be called `UpdateInput`. Its entire definition to read keyboard data looks as follows.

```
void UpdateInput(void)
{
    if(SUCCEEDED(g_Keyboard->Acquire())) //Acquire the device
    {
        char KeyState[256];
        //Take a snapshot
        g_Keyboard->GetDeviceState(sizeof(KeyState), (LPVOID)&KeyState);

        //Question the device
        if(KEYDOWN(KeyState, DIK_LEFT))
            MessageBox(g_hWnd, "You pressed left", "", MB_OK);
    }
}
```

**TIP**

To read keys other than `DIK_LEFT` in the example, view the whole range of keyboard device constants in the DirectX SDK. These are listed under `DirectInput/DirectInput C/C++ Reference/Device Constants/Keyboard Device`. This topic includes a list of others such as `DIK_UP`, `DIK_DOWN`, `DIK_RIGHT`, etc.

## 4.7 The Mouse

The following subsections explain how to create a mouse device, as well as setting the cursor, processing the cursor position, and reading mouse buttons.

### 4.7.1 Creating the Mouse

Having implemented a working keyboard using `DirectInput`, an application can now prepare to add a mouse. This poses a few new challenges to a developer. As expected, creation of the mouse begins much like creation of a keyboard, by creating a new `DirectInput` device pointer (`g_Mouse`), and setting it to `NULL`. Afterward, the device is initialized to a valid interface by calling `CreateDevice`, as explained in Section 4.6.1, and is configured to several starting properties — device sharing and data types — using the `SetDataFormat` and `SetCooperativeLevel` methods from Section 4.6.2. Like keyboards, this code can be written inside `InitInput`, as follows.

```
//Create a mouse
g_lpDI->CreateDevice(GUID_SysMouse, &g_Mouse, NULL);

//Set the data format
g_Mouse->SetDataFormat(&c_dfDIMouse);

//Configure the cooperative level
g_Mouse->SetCooperativeLevel(g_hWnd, DISCL_EXCLUSIVE | DISCL_FOREGROUND);
```

## 4.7.2 Setting the Cursor

After configuring the mouse device comes an aspect of initialization that was not present in keyboard devices: setting the mouse cursor. The cursor is an image — often an arrow — that represents the mouse’s geometric position on screen, and is of course used to click on or select interface items like buttons and text. DirectX represents the cursor like most other images, using surfaces; these were explained in the previous chapter. Consequently, we begin the process of setting a mouse cursor by creating a DirectX surface and loading onto it an image from a file. This is achieved by using the familiar functions of `CreateOffscreenPlainSurface` and `D3DXLoadSurfaceFromFile`. This can be seen in the following code fragment and is a continuation of `InitInput`.

```
//Image info structure
D3DXIMAGE_INFO ImageInfo;

//Load image information, size, etc.
D3DXGetImageInfoFromFile("C:\\Cursor.jpg", &ImageInfo);

//Create a blank surface of appropriate proportions
g_pd3dDevice->CreateOffscreenPlainSurface(ImageInfo.Width,
                                         ImageInfo.Height,
                                         ImageInfo.Format,
                                         D3DPPOOL_DEFAULT,
                                         &g_MouseCursor,
                                         NULL);

//Load an image from a file onto the newly made surface
D3DXLoadSurfaceFromFile(g_MouseCursor, NULL, NULL, ("C:\\Cursor.jpg",
                                         NULL, D3DX_FILTER_NONE, 0xFF000000, NULL));
```

As soon as an image surface has successfully been initialized as an `IDirect3DSurface9` interface and loaded with valid image data using the code above, it becomes a potential mouse cursor. To complete the entire process though, we need to perform the following three simple steps. These all occur with the previously explained Direct3D graphics device (`IDirect3DDevice9`) and continue to be

coded along with the rest of the device initialization inside `InitInput`:

### 1. Register the surface

Register the surface with DirectX using `IDirect3DDevice9::SetCursorProperties`. This tells DirectX which surface, if any, is going to be the active cursor.

```
g_pd3dDevice->SetCursorProperties(0,0, g_MouseCursor);
```



---

#### NOTE

You'll notice that `SetCursorProperties` requires two positional arguments before the surface is specified as the third. These define the X and Y coordinates of the hot spot position within the cursor itself. This is the cursor's target center, relative from the top-left corner of the image (0,0).

### 2. Set cursor position

Set the cursor to an initial position on screen in terms of X and Y with `IDirect3DDevice9::SetCursorPosition`. The starting position could be (0,0), the top left.

```
g_pd3dDevice->SetCursorPosition(0,0,D3DCURSOR_IMMEDIATE_UPDATE);
```



---

#### NOTE

As we shall see, `SetCursorPosition` is called on every occasion the cursor position must be changed, not just on initialization.

### 3. Set cursor visible

Finally, set the cursor visible status to true using the `IDirect3DDevice9::ShowCursor` method. This makes it visible on screen.

```
g_pd3dDevice->ShowCursor(true);
```

### 4.7.3 Reading from the Mouse

Once the mouse device has been created using `CreateDevice`, and then initialized in terms of setting the cooperative level, data format, and cursor properties, we can proceed to read data from it by taking snapshots of its state in the same way data was read from the keyboard. This process occurs inside the `UpdateInput` procedure and begins by calling the `Acquire` function to ensure that the device can provide us with information at this time. Upon success, we call `IDirectInputDevice8::GetDeviceState` to take a freeze-frame of the device at the time of invocation. The subsequent image is returned in a data structure that is different from the 256-byte array used to hold the keystroke data for keyboards. For a mouse, it takes the form of a `DIMOUSESTATE` structure, which contains fields to store information about button states and cursor positions, and whose declaration and parameters are as follows.

```
struct DIMOUSESTATE
{
    LONG IX;
    LONG IY;
    LONG IZ;
    BYTE rgbButtons[4];
};
```

---

**LONG IX:** The distance the mouse cursor has traveled on the X axis since the last snapshot.

---

**LONG IY:** The distance the mouse cursor has traveled on the Y axis since the last snapshot.

---

**LONG IZ:** The distance the mouse cursor has traveled on the Z axis since the last snapshot. This refers to the mouse's scroll wheel or additional scrolling instrument if one exists; if not, this value is consistently 0.

---

**BYTE rgbButtons[4]:** An array of bytes representing a total of four possible buttons on the mouse. If the high-order bit is set, the corresponding button is pressed. As we shall see, this sequence of bytes can be read using the same `KEYDOWN #define` macro for keyboard keys, as explained in Section 4.6.3.

---

Consequently, UpdateInput can be written to acquire the mouse and take a snapshot of it using the following code. Afterward, we process the returned mouse button and mouse cursor data according to the steps illustrated in the following two sections.

```
//Acquire the mouse
if(SUCCEEDED(g_Mouse->Acquire()))
{

//Mouse information structure
DIMOUSESTATE State;

//Populate mouse info struct with snapshot data
g_Mouse->GetDeviceState(sizeof(DIMOUSESTATE), (LPVOID)&State);

}
```

#### 4.7.4 Processing the Cursor Position

Reading the cursor position and updating it visually on screen according to user mouse movement is a simple process. It begins by reading the cursor position from the IX and IY properties of the DIMOUSESTATE structure as it is returned from the GetDeviceState method. These are the standard X and Y values. However, they do not represent the absolute position in terms of the area of the screen. Instead, they describe the *relative* position of the mouse since the last snapshot was taken — in other words, the distance the mouse cursor has traveled on the X and Y axes since GetDeviceState was last called. Therefore, to compute the absolute position of the cursor within the confines of the screen, we must declare two further LONG variables — probably as global members — to maintain an accumulative total of the mouse movements as they're read on each snapshot. This process can be coded as a continuation of Update Input as follows (g\_MouseX and g\_MouseY are the global accumulators, and State is the snapshot data returned from GetDeviceState):

```
g_MouseX += State.lX;
g_MouseY += State.lY;
```

After computing the absolute cursor position, relative to the origin of the screen at the top-left corner (0,0), and storing this value inside a global accumulator that I've named `g_MouseX` and `g_MouseY`, we can complete our processing of the mouse cursor by finally drawing it at the updated position, reflective of user mouse movement. This is achieved by calling the `SetCursorPosition` function of `IDirect3DDevice9`, as follows:

```
g_pd3dDevice->SetCursorPosition(g_MouseX, g_MouseY,
                                D3DCURSOR_IMMEDIATE_UPDATE);
```

## 4.7.5 Reading Mouse Buttons

The final stage of programming the mouse in DirectInput is reading the button information from the mouse device's snapshot, whether they are pressed or whether they are released. This is also the simplest stage. We achieve this by employing the standard `KEYDOWN` `#define` macro that we used for reading keyboard data in Section 4.6.3 to read the pressed status of mouse buttons in the `rgbButtons` array. This is stored in the snapshot structure of `DIMOUSESTATE`. There are four bytes in this array, and each index corresponds to a specific mouse button, where 0 is the leftmost, 3 is the rightmost, and 1 and 2 are those in between. The following code fragment is the complete mouse reading code from `UpdateInput`, with the relevant button reading statement highlighted in bold.

```
if(SUCCEEDED(g_Mouse->Acquire()))
{
    DIMOUSESTATE State;
    g_Mouse->GetDeviceState(sizeof(DIMOUSESTATE), (LPVOID)&State);

    g_MouseX += State.lX;
    g_MouseY += State.lY;
```

```
g_pd3dDevice->SetCursorPosition(g_MouseX, g_MouseY,  
                                D3DCURSOR_IMMEDIATE_UPDATE);  
  
if(KEYDOWN(State.rgbButtons,0))  
    MessageBox(g_hWnd,"You pressed the left button","",MB_OK);  
}
```

## 4.8 Conclusion

---

This chapter and the previous one have covered extensive ground considering the technological breadth of information that's been presented. The next two chapters are about refining this knowledge into a more sophisticated structure. Before moving on, however, let's recap several of the key issues we have learned here:

- A DirectInput object (IDirectInput8) is the topmost manager of all DirectInput interfaces, and is the parent from which all others spring, predominantly through the CreateDevice method.
- DirectInput devices are represented by the IDirectInputDevice8 interface and are created by calling IDirectInput8::CreateDevice. They represent a single input device, such as a keyboard, a mouse, or a joystick. One per device is needed.
- After devices are created, a programmer sets their starting properties, such as cooperative level, data format, and mouse cursor, and then acquires the device on each input cycle to read user input from it. This is achieved by taking a snapshot of the device using GetDeviceState and then questioning the returned data.

# Wrapping Direct3D

This chapter concentrates on utilizing the power of Direct3D and specifically on how it can be modularized into a small collection of reusable classes. These classes are generically termed *wrappers*, and each of them encapsulates a specific component of Direct3D. Particularly, these components are surfaces, textures, and the ID3DXSprite interface. These classes will form a basic graphics library and will incorporate almost everything we'll need to display a user interface. Doing this makes our source code cleaner, simpler, and more abbreviated, and this in turn will make our projects easier to debug. These three classes are described below, and later sections explain each class in more detail.

The CXSurface class encapsulates a single Direct3D surface. It does this by maintaining a valid pointer to the IDirect3DSurface9 interface, and by exposing a number of public methods to perform common operations. These consist of loading images from files, representing the back buffer, and copying data to and from other surfaces as well as drawing them on-screen. See Section 5.1.

Direct3D textures are encapsulated by class CXTexture. Applications should instantiate one instance of this class per texture. See Section 5.2.

Finally, there is CXPen. This encapsulates ID3DXSprite. Practically, this works very much like a GDI pen — or even a normal pen — in that it paints pixels onto the back buffer. It operates hand in hand with textures, and its purpose is to ultimately render one or more instances of CXTexture onto the screen. Typically,

applications that use textures, no matter how many, will only ever need to create one instance of this class. See Section 5.3.

**NOTE**

Please see Book Code\Part I\Chapter5 on the companion CD to view the classes developed in this chapter.

## 5.1 CXSurface — Wrapping Surfaces

The first component to be wrapped in this chapter is surfaces. Chapter 3 explained that surfaces are exactly what their name implies. Geometrically, they can be visualized as a flat plane that is axially aligned to the screen. More technically, they can be thought of as a rectangle of bytes in memory into which images from files like JPEGs and bitmaps can be loaded. Once loaded they're normally drawn to the screen. DirectX represents surfaces using the `IDirect3DSurface9` interface, and it is specifically this interface that is encapsulated into class `CXSurface`. The declaration for the class can be written as below. Following sections discuss how each of these methods and properties will operate.

```
class CXSurface
{
protected:
LPDIRECT3DSURFACE9 m_Surface;    //The surface to be encapsulated
CXSurface* m_BackBuffer;        //Back buffer to be drawn onto
LPDIRECT3DDEVICE9 m_pDevice;    //Direct3D device pointer

public:

CXSurface(LPDIRECT3DDEVICE9 pDevice);
~CXSurface();
LPDIRECT3DSURFACE9 GetSurface() const {return m_Surface;}
void SetSurface(LPDIRECT3DSURFACE9 Surf) const {m_Surface = Surf;}
LPDIRECT3DDEVICE9 GetDevice()const {return m_pDevice;}
void SetDevice(LPDIRECT3DDEVICE9 pDevice) const {m_pDevice = pDevice;}
```

```
HRESULT CreateSurface(UINT Width, UINT Height,  
                      D3DFORMAT Format, D3DPPOOL Pool);  
  
HRESULT LoadFromFile(LPCSTR Path);  
HRESULT MakeBackBuffer(void);  
HRESULT UpdateSurface(CXSurface* Source, RECT* pSourceRect, int X, int Y);  
};
```

## 5.1.1 Instantiating and Deleting CXSurface

As instances of `CXSurface` are created and deleted by an application, its constructor and destructor are called respectively. This class has one constructor function and one destructor function. The constructor ensures the class is initialized correctly, and the destructor ensures deallocation is performed. The definitions of the constructor and destructor are listed below.

```
//This constructor creates a blank surface and initializes a back buffer  
//pointer  
CXSurface::CXSurface(LPDIRECT3DDEVICE9 pDevice)  
{  
    this->SetDevice(pDevice);  
    this->SetSurface(NULL);  
    m_BackBuffer = new CXSurface(pDevice);  
}  
  
//-----  
  
//Destructor deallocates all objects  
CXSurface::~CXSurface()  
{  
    delete m_BackBuffer;  
  
    if(m_Surface != NULL)  
        m_Surface->Release();  
}
```

## 5.1.2 Loading Images

Once an application has declared an instance of `CXSurface`, it will want to load something meaningful onto it. So how does an application actually load image files onto `CXSurface`? Simple — this can be achieved in one swoop by calling the `LoadFromFile` method. This method encapsulates Direct3D's `CreateSurface` and `D3DXLoadSurfaceFromFile` functions all at once. Its definition follows.

```
HRESULT CXSurface::LoadFromFile(LPCSTR Path)
{
    HRESULT Result = E_FAIL;
    D3DXIMAGE_INFO Info;

    ZeroMemory(&Info, sizeof(D3DXIMAGE_INFO));

    if(SUCCEEDED(D3DXGetImageInfoFromFile(Path, &Info)))
    {
        Result = this->CreateSurface(Info.Width, Info.Height, Info.Format,
                                     D3DPPOOL_SYSTEMMEM);
    }

    if(Result == S_OK)
        Result = D3DXLoadSurfaceFromFile(this->m_Surface, NULL, NULL,
                                         Path, NULL, D3DX_FILTER_NONE, 0, NULL);
    else
        Result = E_FAIL;

    return Result;
}
```

## 5.1.3 Copying Surfaces

Some applications may want to copy surfaces. Images stored on `CXSurface` can be copied from one instance to another by using the conventional copy and paste routine. You simply copy a subset of pixels from a source surface and paste them at a specific (X,Y) location on a destination. This process is achieved with the



```
    }  
    else  
        return E_FAIL;  
}
```

## 5.1.5 Rendering

Once a `CXSurface` has been created and initialized it can immediately be drawn to the screen. `CXSurface` makes this process impressively simple. All we need to do is call `CXSurface`'s `Render` method between the `BeginScene` and `EndScene` statements of an application's render loop, and that's it. Consequently, the surface will be drawn to the back buffer, and the result will be shown on-screen as soon as `Present` is called to complete a scene. Let's take a look at the simple function definition for the `Render` method.

```
void CXSurface::Render(void)  
{  
    BackBuffer->UpdateSurface(this,NULL,0,0); //Copy to the back buffer  
}
```

## 5.1.6 Using CXSurface

This section describes how sample applications use the above `CXSurface` class to draw something on-screen. To achieve this, an application would typically perform the following steps: Declare a global pointer to `CXSurface` at the top of its source; initialize this at application start and then load something onto it by calling `LoadFromFile`; call the `Render` method during the render loop to ensure it is drawn to the screen; and finally, delete the surface at application end. The following code demonstrates several of an application's functions that perform these steps.

```
void InitD3D (void)  
{  
    //Create a blank surface  
    g_Surf = new CXSurface(g_pd3dDevice);
```

```

//Load an image from a file
g_Surf->LoadFromFile("Image.jpg");
}

//-----

VOID Render()
{
    // Clear the back buffer to a blue color
    g_pd3dDevice->Clear(0, NULL, D3DCLEAR_TARGET, D3DCOLOR_XRGB(0,0,255),
        1.0f, 0);

    // Begin the scene
    if(SUCCEEDED(g_pd3dDevice->BeginScene()))
    {
        //Render the surface
        g_Surf->Render(); //Render the surface in one simple statement

        // End the scene
        g_pd3dDevice->EndScene();
    }

    // Present the back buffer contents to the display
    g_pd3dDevice->Present(NULL, NULL, NULL, NULL);
}

//-----

void Cleanup()
{
    delete g_Surf;
}

```



#### NOTE

If you don't want to use the Render method of CXSurface, you still can render a surface by explicitly copying it to the back buffer manually. To do this, declare two instances of CXSurface — one instance for a standard surface and the other for the back buffer. Remember to use the GetBackBuffer method. Then, simply copy data between them using UpdateSurface. Such copying should occur during an application's Render procedure, between BeginScene and EndScene.

## 5.2 CTexture — Wrapping Textures

---

The second component in this chapter to be wrapped is textures. Textures are very much like surfaces, except they can perform a number of additional effects. Specifically, they can be drawn on-screen at different positions, scales and orientations. Textures are represented by the `IDirect3DTexture9` interface, and we will encapsulate this into class `CTexture`. Its class declaration is as follows, and its properties and functions are examined in some of the following sections.

```
class CTexture
{
protected:
    LPDIRECT3DTEXTURE9 m_Texture;
    LPDIRECT3DDEVICE9 m_pDevice;
    D3DXVECTOR2 m_RotationCenter;
    D3DXVECTOR2 m_Translation;
    D3DXVECTOR2 m_Scaling;
    FLOAT m_Rotation;
    RECT m_SrcRect;
public:
    CTexture(LPDIRECT3DDEVICE9 pDevice);
    ~CTexture();
    LPDIRECT3DTEXTURE9 GetTexture() const {return m_Texture;}
    void SetTexture(LPDIRECT3DTEXTURE9 Texture) const {m_Texture = Texture;}
    LPDIRECT3DDEVICE9 GetDevice() const {return m_pDevice;}
    void SetDevice(LPDIRECT3DDEVICE9 pDevice) const {m_pDevice = pDevice;}
    D3DXVECTOR2 GetRotationCenter() const {return m_RotationCenter;}
    void SetRotationCenter(D3DXVECTOR2 RotationCenter) {m_RotationCenter =
        RotationCenter;}
    D3DXVECTOR2 GetTranslation() const {return m_Translation;}
    void SetTranslation (D3DXVECTOR2 Translation) const {m_Translation =
        Translation;}
    D3DXVECTOR2 GetScaling() const {return m_Scaling;}
    void SetScaling(D3DXVECTOR2 Scaling) const {m_Scaling = Scaling;}
    FLOAT GetRotation() const {return m_Rotation;}
    void SetRotation (FLOAT Rotation) const {m_Rotation = Rotation;}
    RECT GetRect() const {return m_SrcRect;}
```

```
void SetRect(RECT SrcRect) const {m_SrcRect = SrcRect;}
HRESULT LoadFromFile(char* Path);
};
```

## 5.2.1 Instantiating and Deleting

Textures in terms of `CXTexture` are created and deleted in much the same fashion as `CXSurface`. `CXTexture` has one constructor and one destructor. Inside these functions, the class is initialized and deallocated, respectively. Their function definitions are below.

```
CXTexture::CXTexture(LPDIRECT3DDEVICE9 pDevice)
{
    D3DXVECTOR2 Vec;
    Vec.x = 0;
    Vec.y = 0;
    Vec.z = 0;

    SetDevice(pDevice);
    SetTexture(NULL);
    SetRotation(0.0f);
    SetRotationCenter(Vec);
    SetScaling(Vec);
    SetTranslation(Vec);
}

CXTexture::~CXTexture()
{
    if(m_Texture != NULL)
        m_Texture->Release();
}
```

## 5.2.2 Loading Images

Those who've used `CXSurface` will already have guessed how images are loaded onto `CXTexture`. This is achieved by the `LoadFromFile` method, which encapsulates the `D3DXCreateTextureFromFile` method. Again, this is very simple stuff. `LoadFromFile` looks like this:

```
HRESULT CXTexture::LoadFromFile(char* Path)
{
    if(m_Texture != NULL)
        m_Texture->Release();
    return D3DXCreateTextureFromFile(m_pDevice, Path, &m_Texture);
}
```

### 5.2.3 Preparing to Render

Once an application has loaded and instantiated a `CXTexture` from a file, it will want to position, scale, and orient it in preparation for rendering. These properties are maintained in the `m_Translation`, `m_Scaling`, and `m_Rotation` members. Remember, however, that textures cannot render themselves. To do this, an application will use `CXPen`. This is defined in the next section. The code below demonstrates how a sample application uses `CXTexture` to create a texture and sets its position, scale, and orientation at application startup.

```
void InitD3D (void)
{
    D3DXVECTOR2 Translation;
    D3DXVECTOR2 Scale;

    //Create a blank surface
    g_Texture = new CXTexture(g_pd3dDevice);

    //Load an image from a file
    g_Texture->LoadFromFile("Image.jpg");

    Translation.x = 355;
    Translation.y = 150;
    g_Texture->SetTranslation(Translation);

    Scale.x = 2;
    Scale.y = 2;
    g_Texture->SetScale(Scale);
}
```

## 5.3 CXPen — Wrapping ID3DXSprite

The last Direct3D component in this chapter to be wrapped is the ID3DXSprite interface. Textures cannot draw themselves directly to the back buffer like surfaces can. Instead, they require a pen to be drawn. In DirectX terminology, this pen is known as ID3DX-Sprite. It operates very much like a regular pen. Consequently, we'll encapsulate this interface into a class named CXPen. It will work hand in hand with textures, and its purpose is to draw one or more instances of CXTexture onto the screen during an application's render procedure. CXPen's class declaration is as follows. Likewise, its properties and functions will be examined as the chapter progresses.

```
class CXPen
{
protected:
    LPD3DXSPRITE m_Sprite;
    LPDIRECT3DDEVICE9 m_pDevice;
public:
    CXPen(LPDIRECT3DDEVICE9 pDevice);
    ~CXPen();
    LPD3DXSPRITE GetSprite() const {return m_Sprite;}
    void SetSprite(LPD3DXSPRITE Sprite) const {m_Sprite = Sprite;}
    LPDIRECT3DDEVICE9 GetDevice() const {return m_pDevice;}
    void SetDevice(LPDIRECT3DDEVICE9 pDevice) const {m_pDevice = pDevice;}
    HRESULT DrawTexture(CXTexture* Texture);
};
```

### 5.3.1 Instantiating and Deleting

CXPen has only one constructor and only one destructor, and its constructor requires only one argument. This means that creating instances of this class is a very simple process. We simply pass it a pointer to the associated 3D device, and nothing else. With this method, instances of ID3DXSprite are created and deleted

appropriately. Definitions of CXPen's constructor and destructor are given below.

```
CXPen::CXPen(LPDIRECT3DDEVICE9 pDevice)
{
    this->SetDevice(pDevice);
    D3DXCreateSprite(m_pDevice, &m_Sprite);
}

CXPen::~CXPen()
{
    if(m_Sprite != NULL)
        m_Sprite->Release();
}
```

## 5.3.2 Rendering Textures

For an application to draw a texture on-screen, we need both an instance of CXPen and an instance of CXTexture. Like surfaces, texture drawing should occur between the BeginScene and EndScene statements of an application's rendering loop. The drawing itself is achieved by calling the DrawTexture method of CXPen. This method requires a single argument — a pointer to the texture to render — and then it shall be rendered according to its own position, scale, and orientation settings. (See Section 5.2.3 to learn how to set these values.) Let's take a look at DrawTexture's function definition.

```
HRESULT CXPen::DrawTexture(CXTexture* Texture)
{
    if((Texture != NULL) && (m_Sprite != NULL))
    {
        D3DXMATRIX Mat;
        D3DXMatrixTransformation2D(&Mat, NULL, 0, Texture->GetScaling(),
                                   Texture->GetRotationCenter(), Texture->GetRotation(),
                                   Texture->GetTranslation());

        m_Sprite->Begin(0);
        m_Sprite->SetTransform(&Mat);
    }
}
```

```

        HRESULT Result = m_Sprite->Draw(Texture->GetTexture(),
                                       Texture->GetRect(), NULL, NULL,
                                       0xFFFFFFFF);

        m_Sprite->End();
        return Result;
    }
    else
        return E_FAIL;
}

```

### 5.3.3 Using CXPen and CTexture

The entire Render procedure of an application can now be defined to draw a texture using an instance of CXPen. This can be coded quite simply as follows:

```

VOID Render()
{
    // Clear the back buffer to a blue color
    g_pd3dDevice->Clear(0, NULL, D3DCLEAR_TARGET, D3DCOLOR_XRGB(0,0,255),
                      1.0f, 0);

    // Begin the scene
    if(SUCCEEDED(g_pd3dDevice->BeginScene()))
    {

        g_Pen->DrawTexture(g_Texture);

        // End the scene
        g_pd3dDevice->EndScene();
    }

    // Present the back buffer contents to the display
    g_pd3dDevice->Present(NULL, NULL, NULL, NULL);
}

```

## 5.4 Conclusion

---

This chapter has encapsulated most of Direct3D's capabilities insofar as is relevant to user interfaces. This has taken the form of three reusable classes: `CXSurface`, `CXTexture`, and `CXPen`. And these — together with those developed in the next chapter — compose the rudimentary tools with which we'll build an interface API.

Before moving on I recommend browsing through this chapter once more to review and recap. It is also a good idea to create three or four test applications, some of which use the DirectX API directly and some of which use our classes. This will allow us to appreciate the brevity we're striving for and the clarity this brings.

# Abstracting DirectInput

The previous chapter simplified Direct3D by encapsulating surfaces, textures, and sprites into a series of autonomous wrapper classes. This chapter concentrates on applying the same process to the components of DirectInput. Specifically, we will create separate classes for the DirectInput parent interface, and the keyboard and mouse devices which spring from it. These classes are described below and are explained in more detail throughout the chapter.

The CXInput class will wrap the IDirectInput8 interface, which is known as the DirectInput object. It is also the topmost interface in a DirectInput application from which all others descend, the life-line of DirectInput. CXInput exposes two methods in particular, CreateKeyboard and CreateMouse, and these create instances of DirectInput devices: keyboard (CXKeyboard) and mouse (CXMouse).

CXKeyboard will be returned from the CreateKeyboard method of CXInput. It encapsulates a IDirectInputDevice8 interface, and allows a programmer to read information from a keyboard device. This is accomplished by taking snapshots, or freeze-frames, of the device at regular intervals and then asking questions about its state. Formerly this was achieved with a two-stage process of calling GetDeviceState to take snapshots and then calling the KEYDOWN #define macro to interpret keystroke data. But

CXKeyboard exposes this functionality in the form of the Update and IsKeyPressed methods.

The mouse will be encapsulated by the CXMouse class, and it is instantiated from the CreateMouse method of CXInput. Ultimately — like the keyboard — this class aims to encapsulate a specific variation of the IDirectInputDevice8 interface that manages mouse data. It will expose GetXPos, GetYPos, and IsButtonPressed to read user input. It will also contain methods such as SetCursorPosition and SetCursorVisible to specify various properties about how the device appears to a user.

## 6.1 CXInput — The DirectInput Object

---

Section 4.3 of this book illustrated how a programmer must begin a DirectInput application by calling the DirectInputCreate8 function to create a valid instance of IDirectInput8. This is known as the DirectInput object and is the lifeline of DirectInput. It will be encapsulated into a class called CXInput, and from here we will create mouse and keyboard devices. The declaration for this class is as follows:

```
class CXInput
{
private:
LPDIRECTINPUT8 m_pInput;
  HWND m_hWnd;
protected:
public:
  CXInput (HINSTANCE hinst, HWND hWnd);
  CXKeyboard* CreateKeyboard();
  CXMouse* CreateMouse(LPDIRECT3DDEVICE9 pDevice, bool Exclusive);
  ~ CXInput ();
};
```

## 6.1.1 Instantiating the DirectInput Object

CXInput represents a working instance of the IDirectInput8 interface, which it maintains through its private member: `m_pInput`. In a standard DirectX application, programmers would typically declare pointers to CXInput as a global variable somewhere in their source, and then initialize it inside the `InitInput` procedure or wherever initialization occurs. Object deletion would then occur in procedure `CleanUp` or somewhere close to application close. Examples of initialization and deletion are shown below.

```
//Create in InitInput
g_Input = new CXInput(g_hInst, g_hWnd);

//Delete in CleanUp
if(g_Input != NULL)
    delete g_Input;
```

The constructor will internally create valid instances of the IDirectInput8 interface, and the destructor will release them. The definitions of these two are as follows:

```
//Constructor
CXInput::CXInput (HINSTANCE hinst, HWND hWnd)
{
    m_pInput = NULL;
    m_hWnd = hWnd;

    DirectInput8Create(hinst, DIRECTINPUT_VERSION, IID_IDirectInput8,
        (void*)&m_pInput, NULL);
}

//Destructor
CXInput::~CXInput()
{
    if(m_pInput != NULL)
        m_pInput->Release();
}
```

## 6.1.2 Creating Input Devices

Once `CXInput` is successfully created, we can proceed to create either a mouse or a keyboard device from which user input can be read. This is achieved through two of its public methods: `CreateKeyboard` and `CreateMouse`. These return valid instances of `CXKeyboard` and `CXMouse`, respectively. Their implementations are discussed later in this chapter. These functions are typically called inside `InitInputs`, and instances of the returned classes can be released in `CleanUp`. The process of creation and deletion might look as follows:

```
//InitInputs
g_Keyboard = g_Input->CreateKeyboard();           //Create keyboard device
g_Mouse = g_Input->CreateMouse(g_pd3dDevice, true); //Create mouse device

//Cleanup
delete g_Keyboard;
delete g_Mouse;
```

So the main purpose of `CXInput` is to create devices. `CreateKeyboard` is called to create an instance of a keyboard device, and `CreateMouse` is called for mouse devices. The actual definitions of these functions appear below.

```
//Create keyboard devices
CXKeyboard* CXInput::CreateKeyboard()
{
    CXKeyboard * Keyboard = NULL;

    if(m_pInput)
    {
        Keyboard = new CXKeyboard (m_pInput, m_hwnd);
    }

    return Keyboard;
}
```

```
//-----
//Create mouse devices
CXMuse* CXInput::CreateMouse(LPDIRECT3DDEVICE9 pDevice, bool Exclusive)
{
    CXMouse * Mouse = NULL;

    if(m_pInput)
    {
        Mouse = new CXMouse (pDevice, m_pInput, m_hWnd, Exclusive);
    }

    return Mouse;
}
```

## 6.2 CXKeyboard — Wrapping the Keyboard Device

Keyboard devices are represented by class CXKeyboard. Using this class we can read user input from the keyboard device and set properties about how it behaves. Its class declaration is below.

```
class CXKeyboard
{
private:
    LPDIRECTINPUTDEVICE8 m_pDevice;
    char m_KeyState[256];
protected:
public:
    CXKeyboard (LPDIRECTINPUT8 pInput, HWND hWnd);
    ~ CXKeyboard();
    bool IsKeyPressed(int Key);
    HRESULT Update();
};
```

## 6.2.1 Instantiating Keyboard Devices

Creation of CXKeyboard will occur through the CreateKeyboard method of CXInput, inside an application's InitInput procedure, and deletion of CXKeyboard will occur inside procedure CleanUp, on application close. On these occasions, either the class constructor or the class destructor will be implicitly called. Their definitions are shown below.



### NOTE

The constructor of CXKeyboard accepts two arguments; one is a pointer to the IDirectInput8 interface and the other is a handle to the input focus window. These are used to create a device and immediately initialize it to several starting properties using the SetDataFormat and SetCooperativeLevel methods (see Chapter 4). Essentially, this means that as soon as CXKeyboard is created we have a device that is ready to be used.

```
CXKeyboard::CXKeyboard (LPDIRECTINPUT8 pInput, HWND hWnd)
{
    HRESULT Result = E_FAIL;
    m_pDevice = NULL;

    Result = pInput->CreateDevice(GUID_SysKeyboard, &m_pDevice, NULL);

    if(SUCCEEDED(Result))
    {
        Result = m_pDevice->SetDataFormat(&c_dfDIKeyboard);
        if(FAILED(Result))
        {
            SAFE_RELEASE(m_pDevice);
            return;
        }
    }

    Result = m_pDevice->SetCooperativeLevel (hWnd, DISCL_FOREGROUND |
                                           DISCL_NONEXCLUSIVE);

    if(FAILED(Result))
    {
```

```

        SAFE_RELEASE(m_pDevice);
        return;
    }

    memset(m_KeyState, 0, 256*sizeof(char));

}

//-----

//Destructor
CXKeyboard::~CXKeyboard()
{
    m_pDevice->Unacquire();
    SAFE_RELEASE(m_pDevice);
}

```

## 6.2.2 Reading from CXKeyboard

Keystrokes are read from instances of CXKeyboard through the process of snapshots or freeze-frames. This was explained in Section 4.6.3. Basically, the process involves recording an image of the state of a device at regular enough intervals, such as during Render or UpdateInput. Afterward, we can ask the image a series of questions, such as “Is button A pressed,” in order to determine its state. Using CXKeyboard, snapshots are taken with the Update method, and, based upon its success, questions can subsequently be asked by calling the IsButtonPressed function. An example follows:

**//Checking for a return keypress inside UpdateInput**

```

g_Keyboard->Update(); //Take a snapshot
    if(g_Keyboard->IsKeyPressed(DIK_RETURN)) //Is the Return key pressed?
        MessageBox(g_hWnd, "You pressed Return","",MB_OK); // Then process
// action

```

The definitions for both Update and IsKeyPressed are given below.

```
//Function to take freeze-frames
HRESULT CXKeyboard::Update()
{
    HRESULT Result = E_FAIL;

    if(m_pDevice)
    {
        Result = m_pDevice->Acquire();
        if(FAILED(Result))
            return Result;

        Result = m_pDevice->GetDeviceState(sizeof(m_KeyState),
                                           (LPVOID)&m_KeyState);
        if(FAILED(Result))
            return Result;
    }
    return S_OK;
}

//-----

//Function to read user input from a snapshot
bool CXKeyboard::IsKeyPressed(int Key)
{
    if(m_pDevice)
    {
        if(m_KeyState[Key] & 0x80)
            return true;
    }
    return false;
}
```

## 6.3 Wrapping the Mouse Device

Wrapping the mouse involves the creation of two classes rather than one. This is because there are actually two separate interfaces that we employ when programming the mouse. These are `IDirectInputDevice8` — the device itself — and `IDirect3DSurface9`, a surface for the mouse cursor. Surfaces are a component of `Direct3D` and were wrapped in the previous chapter into class `CXSurface`. There is one instance per surface. So it immediately seems sensible to use this class to encapsulate a cursor. However, this proves to be inappropriate. What if we'll need more than one cursor — say an arrow and an hourglass that we switch between? Or perhaps four or five or even ten cursors? What if we won't initially know how many we'll need and it must change dynamically to suit specific circumstances? Essentially, what we need is a list of `CXSurfaces` that can hold from as little as none to as many cursors as our computer can allow. This concept of a list requires a few changes to be made to `CXSurface` though, and we append these by deriving a new class from it, called `CXMouseSurface`. This issue of a list will be dealt with first before moving on to define `CXMouse` itself.

### 6.3.1 `CXMouseSurface` — Wrapping a List of Cursors

`CXMouseSurface` inherits from `CXSurface`. This means it contains all of the features of its ancestor, while being able to extend its functionality. These features include loading images from files, creating surfaces, and copying them to the back buffer. The small number of additions that have been made in `CXMouseSurface` are highlighted in bold. But only the topmost member is needed to turn it into a list that can grow or shrink as much as required. This list is known as a *linked list* and it is arguably one of the most useful concepts in computer science.

```
class CMouseSurface : public CXSurface
{
private:
    CMouseSurface* m_pNext;
    int MouseSurfaceType;
    UINT HotSpotX;
    UINT HotSpotY;
protected:
public:
    CMouseSurface* GetNext() {return m_pNext;}
    void SetNext(CMouseSurface* Surf) {m_pNext = Surf;}
    int GetSurfaceType() {return MouseSurfaceType;}
    void SetSurfaceType(int Type) {MouseSurfaceType = Type;}
    UINT GetHotSpotX() {return HotSpotX;}
    UINT GetHotSpotY() {return HotSpotY;}
    void SetHotSpotX (UINT X) {HotSpotX = X;}
    void SetHotSpotY (UINT Y) {HotSpotY = Y;}
    CMouseSurface(LPDIRECT3DDEVICE9 pDevice) : CXSurface(pDevice)
                                                {m_pNext = NULL;}
    CMouseSurface() : CXSurface() {m_pNext = NULL;}
};
```

### 6.3.2 Linked Lists — A Definition

A linked list is so called because it represents the concept of a linear list. It spans from left to right, and each item maintains a link to the next. In our case, one instance of `CMouseSurface` is a single item, and it maintains a link to its next item by way of a next pointer. This next pointer is a link to separate working instance of `CMouseSurface`. Figure 6.1 shows an illustration of a normal linked list containing several `CMouseSurface` items.



Figure 6.1

**NOTE**

To some, the idea of a linked list of cursors might seem a little “overkill,” but there are cases when they prove most appropriate. One such instance is graphic adventure games, which include such great titles as *Day of the Tentacle*, *Zork*, *Monkey Island*, and *Tex Murphy*. In these, the player often carries a large inventory of items, such as keys, books, and appliances, and as items are selected the cursor transforms to an iconic representation of it. This indicates it has been chosen. So here is an example most suited to such a cursor arrangement. However, if you’re still inclined to disagree with me, then go ahead and amend the class to use something else — perhaps a fixed size array. Just please keep in mind that linked lists are important and will be used again later in this book.

### 6.3.3 Navigating Linked Lists

Navigation of linked lists occurs linearly, from left to right. Beginning at the leftmost item, we travel iteratively through the entire list from one to the next by way of each item’s next pointer. The end of the list is reached when the next pointer is NULL, indicating that it has no next item. Below is the code to navigate through each item, from start to finish.

```
CXMouseSurface* Temp = this->GetFirstItem(); //Gets the first surface

while(Temp->GetNext()) //Loop all items
{
    //Do something here
    Temp = Temp->GetNext();
}
```

### 6.3.4 Adding New Items to Linked Lists

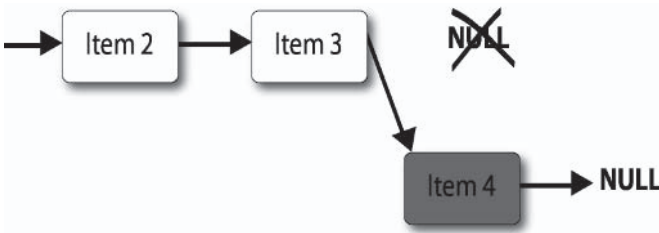


Figure 6.2

Instances of `CXMouseSurface` can be added to the end of the list as new items. For example, to add an item *I* we must first cycle to the list's end, the item with a next pointer of `NULL`. Once there, we change its next pointer to equal the address of *I* and then change the next pointer of *I* to equal `NULL`. This indicates it has now become the last item in the list. The code to achieve this process can be seen below. It will not be stored in `CXMouseSurface` — an individual item — but inside the class that manages the whole list. This will be `CXMouse`, which is defined later in the chapter.

```

void CXMouse::AddCursorSurface(CXMouseSurface* Surface)
{
    if(Surface) //If new item is valid
    {
        if(!this->GetFirstSurface()) //Is the list empty?
            this->m_Surface = Surface; //If so, make it the first item
                                     //in the list
        else
        {
            CXMouseSurface* Temp = this->GetFirstSurface(); //Get first item

            while(Temp->GetNext()) //Cycle through list
                Temp = Temp->GetNext();

            Temp->SetNext(Surface); //Set last item
        }
    }
}

```

### 6.3.5 Deleting Linked Lists

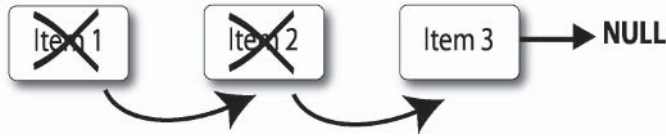


Figure 6.3

A list is cleared when all items are deleted and when the first item equals NULL. This process is achieved by deleting items linearly from left to right. It begins at the first CXMouseSurface. This becomes item *I*. A reference is made to its next item, which becomes known as *N*. Item *I* is then deleted, *N* becomes the new *I*, and its next pointer becomes the new *N*. Again, *I* is deleted and the process repeats until the last item is reached. The code to achieve this, taken from CXMouse, is as follows:

```
CXMouseSurface* I = this->GetFirstSurface(); //Get first item

while(Temp) //Loop all items
{
    CXMouseSurface* N = I->GetNext();
    delete I;
    I = N;
}
```

### 6.3.6 CXMouseSurface — Other Properties

Besides CXMouseSurface having a next pointer that makes it linked list capable, it contains several other additions, most notably HotSpotX and HotSpotY and their corresponding accessor methods. Together, these define the X and Y center of the cursor, whose origin is (0,0), the top-left corner of the cursor surface. As we shall see, these are passed onto the SetCursorProperties method of the Direct3D device, and this occurs inside CXMouse. This is a class that we now define in conjunction with CXMouseSurface.

### 6.3.7 Wrapping the Mouse Device with CXMouse

Instances of CXMouse are returned from the CreateMouse procedure of CXInput. This class allows us to read input from the mouse and to set properties about it. Its declaration can be seen below.

```
class CXMouse
{
private:
    CXMouseSurface* m_Surface; //Pointer to the first item in a cursor list
    CXMouseSurface* m_CurrentCursorSurface; //Pointer to a cursor in the
                                        //list, the selected item
    LPDIRECTINPUTDEVICE8 m_pDevice;      //The mouse device
    LPDIRECT3DDEVICE9 m_p3DDevice;      //The Direct3D device
    DIMOUSESTATE m_State;
    LONG m_iX; //Accumulative (absolute) position of cursor on the X axis
    LONG m_iY; //Accumulative (absolute) position of cursor on the Y axis
protected:
public:
    CXMouse (LPDIRECT3DDEVICE9 pDevice, LPDIRECTINPUT8 pInput, HWND hWnd,
             bool Exclusive);
    ~ CXMouse();
    HRESULT Update();
    LONG GetXPos();
    LONG GetYPos();
    bool IsButtonPressed(int Button);
    HRESULT SetCursorImage();
    HRESULT SetMouseCursor(char* FilePath, UINT HotSpotX, UINT HotSpotY,
                           int Type);
    void AddCursorSurface(CXMouseSurface* Surface);
    CXMouseSurface* GetFirstSurface() {return m_Surface;}
    bool SetCursor(int Type);
    CXMouseSurface* GetCurrentCursor() {return m_CurrentCursorSurface;}
    void SetCurrentCursor(CXMouseSurface* Surface)
        {m_CurrentCursorSurface = Surface;}
    void SetCursorPosition(int X, int Y);
    HRESULT SetCursorVisible(bool Show);
};
```

Creation occurs during an application's InitInput procedure, and deletion occurs inside CleanUp on application end. At these times,

we invoke either the class constructor or destructor. These can be seen below. The constructor creates instances of `IDirectInputDevice8` and sets it to several starting properties, while the destructor destroys such instances.

**//Constructor**

```

CXMouse::CXMouse(LPDIRECT3DDEVICE9 pDevice, LPDIRECTINPUT8 pInput,
                 HWND hWnd, bool Exclusive)
{
    m_p3DDevice = NULL; //Set Direct3D device

    //Initial cursor position
    m_iX = 0;
    m_iY = 0;

    if((pInput) && (pDevice))
    {
        HRESULT Result = E_FAIL;
        m_p3DDevice = pDevice;

        Result = pInput->CreateDevice(GUID_SysMouse, &m_pDevice, NULL);

        if(FAILED(Result))
            return;

        Result = m_pDevice->SetDataFormat(&c_dfDIMouse);

        if(FAILED(Result))
        {
            SAFE_RELEASE(m_pDevice);
            return;
        }

        if(Exclusive)
        {
            Result = m_pDevice->SetCooperativeLevel(hWnd, DISCL_EXCLUSIVE |
                                                    DISCL_NOWINKEY |
                                                    DISCL_FOREGROUND);
        }
        else
        {
    
```

```

        Result = m_pDevice->SetCooperativeLevel(hWnd, DISCL_NONEXCLUSIVE |
                                                DISCL_FOREGROUND);

    }
    if(FAILED(Result))
    {
        SAFE_RELEASE(m_pDevice);
        return;
    }

    ZeroMemory(sizeof(&Mode), (void*)&Mode);
    m_p3DDevice->GetDisplayMode(0,&Mode);

    m_Surface = NULL;
    m_CurrentCursorSurface = NULL;
    m_Changed = false;
    m_Buttons = false;
}
}

//-----

//Destructor
CXMouse::~CXMouse()
{
    if(m_pDevice)
    {
        m_pDevice->Unacquire();
        SAFE_RELEASE(m_pDevice);
    }

    CXMouseSurface* Temp = this->GetFirstSurface();

    while(Temp)
    {
        CXMouseSurface* Next = Temp->GetNext();
        SAFE_DELETE(Temp);
        Temp = Next;
    }
}

```

### 6.3.8 Initializing Mouse Cursors with CXMouse

After CXMouse is created, we must perform an additional step if we want to add a mouse cursor that is going to be visible on-screen. CXMouse maintains a list of cursors, and it achieves this by keeping a pointer to the first item in the list. This item is of type CXMouseSurface\*. When it is NULL, the list is empty. When it's not NULL, the list contains *at least* one cursor. When the first item's next pointer is equal to NULL, the list contains *only* one cursor. When it isn't NULL, it contains more. To add a cursor we must call the SetMouseCursor method of CXMouse. This function accepts a path to the cursor image, X and Y hot spots, and a unique identifier to help select it from the list. Subsequent to this call we must also invoke SetCursorPosition and SetCursorVisible. Examples of these functions setting a mouse cursor are shown below.

```
//InitDeviceObjects
m_Mouse->SetMouseCursor("C:\\ cursor.dds", 0, 0, 0); //Create a cursor
m_Mouse->SetCursorPosition(0,0); //Set starting position
m_Mouse->SetCursorVisible(true); //Is it visible?
```

The definitions of SetMouseCursor, SetCursorPosition, and SetCursorVisible are as follows:

```
//Set mouse cursor
HRESULT CXMouse::SetMouseCursor(char* FilePath, UINT HotSpotX,
                                UINT HotSpotY, int Type)
{
    //Create a cursor – see Sections 5.2 and 6.3.1
    CXMouseSurface* Tmp = new CXMouseSurface(this->m_p3DDevice);

    //Load image from surface – see Sections 5.1.2 and 5.2.2
    HRESULT Result = Tmp->LoadFromFile(FilePath);

    if(SUCCEEDED(Result))
    {
        //Set cursor properties
        Tmp->SetHotSpotX(HotSpotX);
        Tmp->SetHotSpotY(HotSpotY);
    }
}
```

```
        Tmp->SetSurfaceType(Type);

        //Add cursor to end of list – see section 6.3.4
        this->AddCursorSurface(Tmp);

        //Make this the active – currently visible – cursor
        this->SetCurrentCursor(Tmp);

//Finally register the cursor with DirectX

return m_p3DDevice->SetCursorProperties(this->GetCurrentCursor()
                                     ->GetHotSpotX(),
                                     this->GetCurrentCursor()
                                     ->GetHotSpotY(),
                                     this->GetCurrentCursor()
                                     ->GetSurface());
    }

    return Result;
}

//-----

//Set cursor position
void CXMouse::SetCursorPosition(int X, int Y)
{
    m_p3DDevice->SetCursorPosition(X,Y,0);
}

//-----

//Set cursor visible
HRESULT CXMouse::SetCursorVisible(bool Show)
{
    return m_p3DDevice->ShowCursor(Show);
}
```

### 6.3.9 Changing Mouse Cursors with CXMouse

Only one cursor in the list can be the current cursor at any one time, (e.g., you can have either an hourglass or an arrow on-screen but not both). This is known as the active cursor and is represented by the member `m_CurrentCursorSurface`. More often than not, this property is implicitly set as cursors are added using `SetMouseCursor`. But there will be many occasions when you'll want to change the cursor from one to another yourself. In other words, you want to swap the active cursor. This can be done using the `SetCursor` method. All it requires is the unique identifier you assigned to a cursor upon creation (see previous section). An example of this function in use appears below, along with its definition.

```
//Set mouse cursor use
m_Mouse->SetCursor(0);

//-----

//Set mouse cursor definition
bool CXMouse::SetCursor(int Type)
{
    bool Found = false;
    CXMouseSurface* Temp = this->GetFirstSurface();

    while(Temp)
    {
        if(Temp->GetSurfaceType() == Type)
        {
            Found = true;
            this->SetCurrentCursor(Temp);
            m_p3DDevice->SetCursorProperties(this->GetCurrentCursor()
                ->GetHotSpotX(),
                this->GetCurrentCursor()
                ->GetHotSpotY(),
                this->GetCurrentCursor()
                ->GetSurface());
        }
    }
}
```

```

        break;
    }
    Temp = Temp->GetNext();
}
return Found;
}

```

### 6.3.10 Reading Mouse Data with CXMouse

Mouse data, in terms of button states and cursor positions, is read in much the same fashion as keyboard data. A snapshot is taken of the device's state at regular enough intervals using `UpdateInput`, and then questions are asked about it. In this case, those questions will relate to the X and Y position of the mouse cursor and the depressed status of the mouse's buttons. To first take a snapshot of the device we call the `Update` method. The definition of this method looks like this:

```

HRESULT CXMouse::Update()
{
    HRESULT Result = E_FAIL;
    long OldX, OldY;
    bool Pressed = this->IsButtonPressed(0);
    m_Changed = false;

    if(m_pDevice)
    {
        Result = m_pDevice->Acquire();
        Result = m_pDevice->Poll();

        if(FAILED(Result))
            return Result;

        Result = m_pDevice->GetDeviceState(sizeof(DIMOUSESTATE), (void*)
            &m_State);

        if(FAILED(Result))
            return Result;
    }
}

```

```

        if(this->IsButtonPressed(0) != Pressed)
            m_Buttons = true;
        else
            m_Buttons = false;

        m_iX += m_State.lX;
        m_iY += m_State.lY;

    }

    return Result;
}

```

### 6.3.11 Reading Cursor Positions with CXMouse

Update is called to take snapshots of a device, and afterward cursor positions can be read using the GetXPos and GetYPos functions. These properties record the accumulative position of the mouse — its absolute position. This is relative to the top-left corner of the screen. An example of reading the cursor position is presented below. It could be coded inside UpdateInput or in a Render procedure.

```

m_Mouse->Update();                //Take snapshot

long X = m_Mouse->GetXPos();      //Get X pos
long Y = m_Mouse->GetYPos();      //Get Y pos

m_Mouse->SetCursorPosition(X, Y); //Update cursor position on screen

```

### 6.3.12 Reading Button States with CXMouse

Like cursor positions, button states are read after a snapshot has been taken using the Update function. These can be read with IsButtonPressed. An example of this appears below. Again, it can be coded inside UpdateInput or Render.

```
m_Mouse->Update();                //Take snapshot

if(m_Mouse->IsButtonPressed(0)) //Is button pressed?
    MessageBox(this->m_hwnd, "You pressed left","",MB_OK); //Respond
```

## 6.4 Conclusion

---

This chapter marks the end of Part I of this book. It brings to a close a rounded knowledge of Direct3D and DirectInput. Before moving on, it is advised that you take a look back through Part I and examine the code on the CD, specifically examining the wrappers we have created throughout the past two chapters. These constitute our tools for the programming tasks ahead. It is by using these that we will construct a reusable interface library, complete with windows, controls, and more.

# Beginning CXControl

This is the beginning of Part II. Here, the knowledge from Part I is used to develop a reusable interface library, called UI LIB. Later in this book, UI LIB will be deployed to create an interface for a DirectShow Media Player program. However, this chapter concentrates on creating the beginnings of that library. In doing so it answers the following questions:

- What is an interface library?
- How do we develop controls?
- What is a base control?
- How does a control hierarchy work?
- Which properties are common to all controls?
- How can we implement parent, child, and sibling controls?
- What is a canvas?
- What is relative positioning?

## 7.1 UI LIB (User Interface Library) — What Is It?

---

Programmers are always on the lookout for shortcuts, as none of us wants to reinvent the wheel. Consequently, when developing applications we tend to include different libraries containing premade functions to perform the hard work for us. For example, file I/O functions or `printf` and `scanf` routines allow us to achieve different tasks without having to learn the specifics of hardware. Thus, using libraries saves us time and makes our software easier to support. UI LIB, which we develop in Part II, will provide developers those same benefits in the context of developing interfaces. Ultimately, it will contain a collection of classes and functions that will assemble first-rate interfaces in a short period of time. For developers using our library, it will be as simple as typing `#include <UILIB.H>` in their projects, and including its appropriate lib, for our library's entire interface benefits to become available. Let us now examine UI LIB more closely and see what it will consist of.

## 7.2 UI LIB — Controls as Classes

---

Chapter 1 explained how interfaces are made from a collection of controls, like buttons, list boxes, text boxes, check boxes, etc; and it also explained how both a user and program use controls to communicate. Therefore, UI LIB shall be a collection of controls. Fortunately for us, controls can be encapsulated into classes — one class per control, like class `CButton`, `CTextBox`, `CListBox`, and so on. Hence, UI LIB shall essentially be a collection of classes, and those classes will be controls.

## 7.3 Controls — Class Hierarchy and Base Controls

So where is the best place to start developing UI LIB? Should we begin by developing a button control? A text box control? Or even a drop-down list control? Does it really matter, or is there some basic developmental structure to which we should adhere? Indeed, the best place to start is by asking ourselves exactly what a control is. It is only by doing this that we can understand that all controls, regardless of their differences, share a common set of attributes. In fact, it is only by having at least these attributes that they may be considered a control. These attributes are examined as the chapter progresses. In terms of classes then, it makes sense to develop our controls in stages, beginning with a class that encapsulates everything all controls share, and nothing more. This is known as a base class, or base control. It is not typically instantiated alone, but whenever other controls are created, such as buttons or labels, they'll be derived from it. Hence, those descendants become derived classes, or derived controls, because they inherit and extend upon the features of their ancestor. This will save us having to code the same functionality into each control individually. This base class will be called CXControl, and it forms the subject matter for the rest of this chapter and the next. Figure 7.1 demonstrates the class hierarchy for UI LIB. Note that CXControl is highlighted.

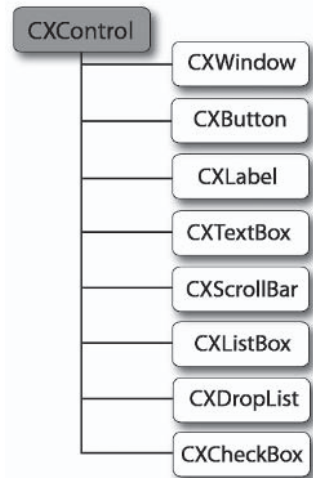


Figure 7.1



---

**NOTE**

CXControl spans two chapters because it's so large and incorporates important concepts. I have coded a lot of power into this class so that customizing derived classes will be that much quicker and simpler.

## 7.4 CXControl — The Beginnings

---

CXControl is the ancestor upon which other classes inherit a common feature set. It is therefore a base class, and other controls in UI LIB are derived classes. The rest of this chapter is dedicated to the development of CXControl. Specifically, subsequent sections investigate features common to all controls and then implement them into CXControl for descendants to inherit. That development process begins here, with a blank class declaration, and will be added to as the chapter progresses.

```
class CXControl
{
private:
protected:
public:
};
```



---

**NOTE**

CXControl is being developed inside BaseControl.h. You can follow through this chapter and code it from scratch or load the file from the companion CD. This can be found at Book Code\Part II\Chapters 7 and 8\BaseControl.h.

## 7.5 Defining CXControl — Controls and a Canvas

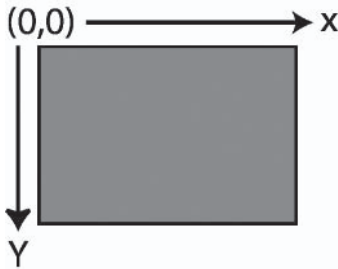


Figure 7.2. A canvas before it is drawn upon.

Visually, controls differ widely; list boxes have one appearance and buttons have another. However, all controls typically represent a rectangular region inside the boundaries of their parent, and inside this region a control will paint itself. Appropriately enough, this rectangular drawing area is termed a *canvas*. Technically, it is a collection of pixels — analogous to a surface or a texture. Its size can be expressed in terms of width and height — measured in pixels — and its visibility status can be either visible or invisible. Take a look at CXControl’s following declaration to see how a canvas has successfully been implemented.

```
class CXControl
{
protected:
    DWORD m_Width;           //Width of canvas
    DWORD m_Height;         //Height of canvas
    bool m_Visible;         //Visibility of canvas
    CXTexture* m_Canvas;    //The canvas itself
    CXPen* m_Pen;           //Something to draw upon the canvas
public:
    CXTexture* GetCanvas(void) {return m_Canvas;}
    void SetCanvas(CXTexture* Texture) {m_Canvas = Texture;}
    bool GetVisible(void) {return m_Visible;}
```

```
void SetVisible(bool Visible) {m_Visible = Visible;}
CXPen* GetPen(void) {return m_Pen;}
void SetPen(CXPen* Pen) {m_Pen = Pen;}
DWORD GetWidth(void) {return m_Width;}
DWORD GetHeight(void) {return m_Height;}
void SetWidth(DWORD Width) {m_Width = Width;}
void SetHeight(DWORD Height) {m_Height = Height;}
};
```



---

**NOTE**

The details of drawing are defined in the next chapter as we continue to develop CXControl, and specifically as we investigate messages and event handling.



---

**NOTE**

Nowadays, in a world of skinnable apps like media players and instant messengers, users can customize the look of their controls such that they are non-rectangular. This can be implemented in Direct3D by using alpha tested textures. However, this is an advanced topic that is beyond the scope of this book.

## 7.6 CXControl — Parent, Sibling, and Child Controls

---

Chapter 1 explained briefly how every control of an interface exists inside a hierarchy. Hence, controls are intimately related. For example, the topmost control of an interface has no parent, except the desktop. More often than not, this control is the main application window, and this will contain other controls like buttons and check boxes, etc. Such controls are therefore children of a window, siblings to one another, and the window itself is their parent. In fact, this hierarchy happens to be one of the most important influences on a control, determining both when they're created and destroyed in an application, among other things. See Figure 7.3 to visualize how this hierarchical relationship could appear diagrammatically.

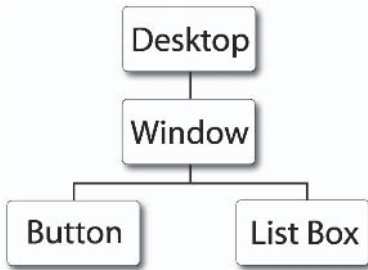


Figure 7.3

In the previously chapter, we examined how a list of mouse cursors could be managed effectively by linked lists. We will now employ an improvement of that process here to handle a control's relationship. Recall that a linked list is a linear list of items where each item maintains a pointer to its next, except for the last item, whose next is NULL. Typically, this is an ideal arrangement to store lists of items, like children of a control. However, the problem is that you can only traverse a linked list linearly from left to right, one to the next. Now, while this isn't inherently a catastrophic obstacle, it can often be an inconvenient and impractical structure. The solution, therefore, is to extend the linked list concept to become a two-way linked list. Here, each control maintains a pointer to both its next and previous siblings; in other words, pointers to both the control before and the control after any current item in the list. For a developer, such an arrangement provides several unique benefits: One, you can navigate the list in a bidirectional way, reaching either end from any starting point; two, you can delete arbitrary items and repair residual holes in the chain; and three, you can perform all kinds of sorting and item rearrangement operations. See Figure 7.4 to get an idea of how two-way linked lists appear.



Figure 7.4

Thus, to implement parent, child, and sibling relationships in controls, it makes sense to use a two-way linked list. Such a list can be appended to class `CXControl` to handle its children simply by adding several different pointers: one to a parent control, one to the next and previous siblings, and one to a control's first child. To manage these pointers, a number of functions should be added. These include functions to add child controls, functions to navigate through lists of siblings, and functions to delete child controls. These are explained in some of the following subsections. Take a look at the appended class declaration for `CXControl` below.

```

class CXControl
{
protected::
  DWORD m_Width;
  DWORD m_Height;
  bool m_Visible;
  CTexture* m_Canvas;
  CXPen* m_Pen;
  CXControl* m_ChildControls;
  CXControl* m_NextSibling;
  CXControl* m_PreviousSibling;
  CXControl* m_Parent;
public:
  //Accessors
  CTexture* GetCanvas(void) {return m_Canvas;}
  void SetCanvas(CTexture* Texture) {m_Canvas = Texture;}
  bool GetVisible(void) {return m_Visible;}
  void SetVisible(bool Visible) {m_Visible = Visible;}
  CXPen* GetPen(void) {return m_Pen;}
  void SetPen(CXPen* Pen) {m_Pen = Pen;}
  DWORD GetWidth(void) {return m_Width;}
  DWORD GetHeight(void) {return m_Height;}

```

```

void SetWidth(DWORD Width) {m_Width = Width;}
void SetHeight(DWORD Height) {m_Height = Height;}
CXControl* GetParentControl(void) {return m_Parent;}
void SetParentControl(CXControl* Control) {m_Parent = Control;}
CXControl* GetNextSibling(void) {return m_NextSibling;}
void SetNextSibling(CXControl* Control) {m_NextSibling = Control;}
CXControl* GetPreviousSibling(void) {return m_PreviousSibling;}
void SetPreviousSibling(CXControl* Control) {m_PreviousSibling = Control;}
CXControl* GetFirstChild(void) {return m_ChildControls;}
void SetFirstChild(CXControl* Control) {m_ChildControls = Control;}

CXControl* AddChildControl(CXControl* Control);
CXControl* RemoveChildControl(CXControl* Control);
void RemoveAllChildren();
int GetChildCount();
};

```

## 7.6.1 Adding Child Controls

Controls store references to their children by way of the `m_ChildControls` pointer, and this effectively represents the first child in a list of potentially many. Thus, to make an existing control a child of another, one must call the `AddChildControl` method of `CXControl`. For example, one would use this function to make a button control a child of a window. Examine the function definition below. You saw a similar function in the previous chapter to add cursors to a list.

```

CXControl* CXControl::AddChildControl(CXControl* Control)
{
    Control->SetParentControl(this);
    CXPen* Pen = Control->GetPen();
    SAFE_DELETE(Pen);
    Control->SetPen(this->GetPen());

    if(!m_ChildControls)
        m_ChildControls = Control;
    else
    {

```

```

        CXControl* Temp = this->GetFirstChild();

        while(Temp->GetNextSibling())
            Temp = Temp->GetNextSibling();

        Temp->SetNextSibling(Control);
        Control->SetPreviousSibling(Temp);
    }

    return Control;
}

```




---

**NOTE**

Notice that this function differs somewhat from the previous chapter where we added cursors to a linked list. Here, we're creating a two-way linked list and are therefore setting the previous sibling control in addition to the next. Hence, this now means we can navigate a list of child controls in a bidirectional way.

## 7.6.2 Clearing Child Controls

Clearing child controls is the process of deleting them all. To do this, you must call the `RemoveAllChildren` method of `CXControl`. Again, a similar process was demonstrated in the previous chapter. Let's take a look at its function definition.

```

void CXControl::RemoveAllChildren()
{
    CXControl* Temp = this->GetFirstChild();

    while(Temp)
    {
        CXControl* Next = Temp->GetNextSibling();
        SAFE_DELETE(Temp);
        Temp = Next;
    }
}

```

### 7.6.3 Removing Specific Children

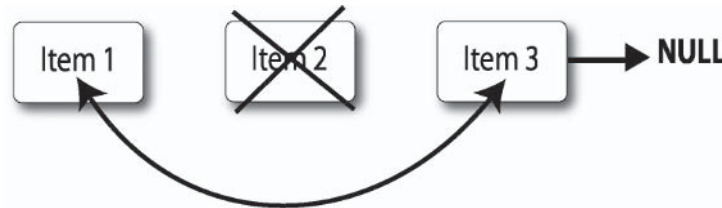


Figure 7.5

Something we didn't see in the previous chapter was the deleting of arbitrary items that could appear anywhere in the list. However, two-way linked lists make this process simple. For instance, to delete an item *I*, perform the following steps: Keep a reference *N* to the next sibling and keep a reference *P* to previous sibling. Then delete *I*. Finally, change the next sibling of *P* to *N*. See Figure 7.5 and the following definition of the `RemoveChildControl` method of `CXControl`.

```
CXControl* CXControl::RemoveChildControl(CXControl* Control)
{
    CXControl* Next = Control->GetNextSibling();
    CXControl* Previous = Control->GetPreviousSibling();
    SAFE_DELETE(Control);
    Next->SetPreviousSibling(Previous);
    Control = Next;
    return Control;
}
```

## 7.6.4 Counting Child Controls

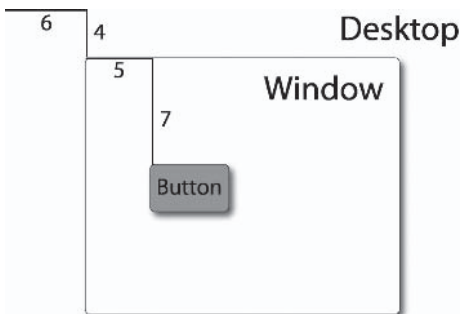
Occasionally it will be useful to know how many children a specific control has. Counting them is a simple process. You must iteratively traverse the list of children from left to right and increment a counter as you proceed. This process is achieved by calling the `GetChildCount` method of `CXControl`. Its definition follows.

```
int CXControl::GetChildCount()
{
    int Count = 0;
    CXControl* Temp = this->GetFirstChild();

    while(Temp)
    {
        CXControl* Next = Temp->GetNextSibling();
        Count++;
        Temp = Next;
    }
    return Count;
}
```

## 7.7 Absolute and Relative Positioning

---



*Figure 7.6. Notice that the buttons' relative and absolute positions are not the same. They describe different values. One represents its position on screen, and the other represents its position inside its parent control.*

Section 7.5 explained how a canvas is, in essence, everything that is visible about a control. It demonstrated how a control's size is expressed in terms of width and height, and its visibility status in terms of visible and invisible. However, there's an additional property thus far overlooked, that of position. Specifically, a control has two positions in terms of X and Y. These are its absolute position and its relative position. Absolute position is the idea that immediately springs to mind when people think of a position; it essentially defines a control's X and Y location, measured from the top-left corner of the screen. Its relative position, however, defines a control's X and Y position relative to its *parent* — in other words, measured from the top-left corner of its parent, not the top-left corner of the screen. Invariably, some people might be tempted to question whether such a distinction matters, but the answer would be a resounding “Yes.” To see why, take a look at Figure 7.6. For the purposes of UI LIB, all controls should maintain their relative positions rather than absolute positions because it's simpler and more intuitive. However, it will often be necessary to compute their absolute positions too, and we'll see how this is achieved in the next section.

```
class CXControl
{
protected:
D3DXVECTOR2 m_Position;
    DWORD m_Width;
    DWORD m_Height;
    bool m_Visible;
    CTexture* m_Canvas;
    CXPen* m_Pen;
    CXControl* m_ChildControls;
    CXControl* m_NextSibling;
    CXControl* m_PreviousSibling;
    CXControl* m_Parent;
public:
    //Accessors
    CTexture* GetCanvas(void) {return m_Canvas;}
    void SetCanvas(CTexture* Texture) {m_Canvas = Texture;}
    bool GetVisible(void) {return m_Visible;}
```

```
void SetVisible(bool Visible) {m_Visible = Visible;}
CXPen* GetPen(void) {return m_Pen;}
void SetPen(CXPen* Pen) {m_Pen = Pen;}
DWORD GetWidth(void) {return m_Width;}
DWORD GetHeight(void) {return m_Height;}
void SetWidth(DWORD Width) {m_Width = Width;}
void SetHeight(DWORD Height) {m_Height = Height;}
CXControl* GetParentControl(void) {return m_Parent;}
void SetParentControl(CXControl* Control) {m_Parent = Control;}
CXControl* GetNextSibling(void) {return m_NextSibling;}
void SetNextSibling(CXControl* Control) {m_NextSibling = Control;}
CXControl* GetPreviousSibling(void) {return m_PreviousSibling;}
void SetPreviousSibling(CXControl* Control) {m_PreviousSibling = Control;}
CXControl* GetFirstChild(void) {return m_ChildControls;}
void SetFirstChild(CXControl* Control) {m_ChildControls = Control;}
D3DXVECTOR2* GetPosition(void) {return &m_Position;}
FLOAT GetXPos(void) {return m_Position.x;}
FLOAT GetYPos(void) {return m_Position.y;}
void SetXPos(FLOAT X) {m_Position.x = X;}
void SetYPos(FLOAT Y) {m_Position.y = Y;}
void SetXYPos(FLOAT X, FLOAT Y);

CXControl* AddChildControl(CXControl* Control);
CXControl* RemoveChildControl(CXControl* Control);
void RemoveAllChildren();
int GetChildCount();
void GetAbsolutePosition(D3DXVECTOR2* Position);
};
```



---

**NOTE**

For the topmost control in a hierarchy, such as the main application window, the absolute and relative positions are the same. This is because topmost controls have no parent, except the desktop, whose extent spans the whole of the screen.



---

**NOTE**

When dealing with the Windows API, these positions are often referred to as screen coordinates, or relative and parent coordinates.

## 7.7.1 Computing Positions

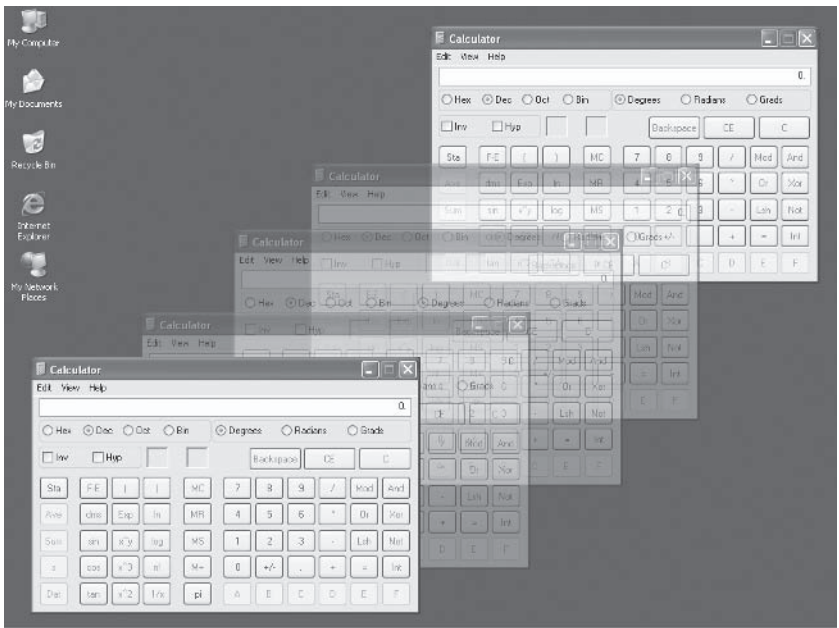


Figure 7.7. By maintaining a control’s relative position, it’s easy to compute its absolute position. This makes it simple to redraw controls correctly when a window is dragged across the screen, such that each control maintains the same relative distance from the other.

The `GetAbsolutePosition` method of `CXControl` returns a control’s absolute position — in other words, its actual X and Y location on the screen. We will most often use absolute positions to paint a control correctly on-screen. Computing the *absolute* position for a control is a simple process: You simply add the control’s *relative* position to its parent’s *absolute* position. Essentially, this gives you the cumulative total of all relative positions, from one control to its topmost parent in a hierarchy. Take a look at the function definition for `GetAbsolutePosition`.

```
void CXControl::GetAbsolutePosition(D3DXVECTOR2* Position)
{
    Position->x += this->GetXPos();
    Position->y += this->GetYPos();

    if(this->m_Parent)
        this->m_Parent->GetAbsolutePosition(Position);
}
```

## 7.8 CXControl — The Class Declaration Thus Far

---

```
class CXControl
{
protected:
    D3DXVECTOR2 m_Position;
    DWORD m_Width;
    DWORD m_Height;
    bool m_Visible;
    CTexture* m_Canvas;
    CXPen* m_Pen;
    CXControl* m_ChildControls;
    CXControl* m_NextSibling;
    CXControl* m_PreviousSibling;
    CXControl* m_Parent;
public:
    //Accessors
    CTexture* GetCanvas(void) {return m_Canvas;}
    void SetCanvas(CTexture* Texture) {m_Canvas = Texture;}
    bool GetVisible(void) {return m_Visible;}
    void SetVisible(bool Visible) {m_Visible = Visible;}
    CXPen* GetPen(void) {return m_Pen;}
    void SetPen(CXPen* Pen) {m_Pen = Pen;}
    DWORD GetWidth(void) {return m_Width;}
    DWORD GetHeight(void) {return m_Height;}
    void SetWidth(DWORD Width) {m_Width = Width;}
    void SetHeight(DWORD Height) {m_Height = Height;}
```

```

CXControl* GetParentControl(void) {return m_Parent;}
void SetParentControl(CXControl* Control) {m_Parent = Control;}
CXControl* GetNextSibling(void) {return m_NextSibling;}
void SetNextSibling(CXControl* Control) {m_NextSibling = Control;}
CXControl* GetPreviousSibling(void) {return m_PreviousSibling;}
void SetPreviousSibling(CXControl* Control) {m_PreviousSibling = Control;}
CXControl* GetFirstChild(void) {return m_ChildControls;}
void SetFirstChild(CXControl* Control) {m_ChildControls = Control;}
D3DXVECTOR2* GetPosition(void) {return &m_Position;}
FLOAT GetXPos(void) {return m_Position.x;}
FLOAT GetYPos(void) {return m_Position.y;}
void SetXPos(FLOAT X) {m_Position.x = X;}
void SetYPos(FLOAT Y) {m_Position.y = Y;}
void SetXYPos(FLOAT X, FLOAT Y);
CXControl* AddChildControl(CXControl* Control);
CXControl* RemoveChildControl(CXControl* Control);
void RemoveAllChildren();
int GetChildCount();
void GetAbsolutePosition(D3DXVECTOR2* Position);
};
    
```

## 7.9 Conclusion

This chapter presented the beginnings of UI LIB, which took the form of class CXControl. This class encapsulates attributes that are common to all controls, features all of them must possess in order to be a control. The next chapter continues this work. However, before moving on, let's recap what this chapter has demonstrated.

- A library is a collection of functions, structures, and classes that achieves specific tasks. Libraries are primarily designed to make a developer's life simpler by providing him with the tools to the get job done. DirectX is an example of a library.
- UI LIB is an abbreviation for user interface library. It will consist of a series of controls — such as buttons, list boxes, check boxes, etc. — that allows developers to construct user interfaces for their software.

- Even though controls differ, they all inherit a common set of attributes. This is why CXControl has been developed. It is not typically instantiated on its own, but acts as a base control from which others derive a basic feature set.
- Geometrically, controls are typically a rectangular region called a canvas. This can be expressed in terms of width and height, and visible and invisible. Inside this canvas a control draws itself to a user; hence, buttons have one appearance and lists have another.
- Each control in an interface exists in a hierarchy. The topmost control is known as the ultimate ancestor — or root control — and this is often the main window. Other controls are descendants of this, and these too can have siblings and children.
- Controls have two positions, an absolute position and a relative position. The former expresses a control's actual position from the top-left corner of the screen, in terms of X and Y. The latter, however, expresses a control's position from the top-left corner of its parent, not the screen. This process is useful for readjusting controls whenever a window is dragged across the screen.

# Continuing CXControl

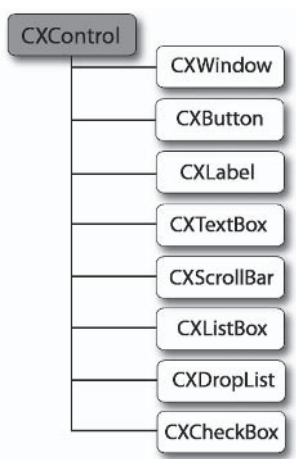


Figure 8.1

The whole of Part II is dedicated to developing UI LIB, a collection of controls. This began in the previous chapter by developing a base class defining the common attributes of controls that are inherited by descendant classes. Descendants are controls such as buttons, list boxes, check boxes, and even windows. These will be developed in subsequent chapters. This chapter, however, continues where the previous left off, by completing the base class CXControl. In this chapter, the following topics are explored:

- Event handling
- Windows messages and custom messages
- Posting and processing messages
- Painting
- Depth sorting
- Focus

**TIP**

You can resume coding from the previous chapter or load this chapter's work from the CD at Book Code\Part II\Chapters 7 and 8.

## 8.1 Messages

For controls to operate properly they need to know when specific events occur. For example, text boxes must respond to keypresses and buttons must respond to mouse clicks, etc. The way an application receives events is through the `WndProc` procedure, part of the message pump. Here, events are received in the form of messages. These are actually data structures passed as arguments to `WndProc` whenever events occur. Effectively, they are data packets whose properties describe information about an event, such as which key was pressed or the new location of the mouse cursor. The following declaration shows a windows message, and Table 8.1 lists the most common messages and describes their properties.

```
struct Message
{
    UINT Message;           //The type of message; mouse message, etc. See
                           //Table 8.1 for a selective list
    LPARAM Parameter1;     //Describes additional information about an event
    WPARAM Parameter1;     //Describes additional information about an event
};
```

**Table 8.1. Common messages**

<p><b>WM_PAINT</b></p>	<p>Sent to an application whenever a window needs to be redrawn; more specifically, whenever the GDI identifies a window or its contents need to be drawn in whole or part. However, it should be remembered that Direct3D applications do not rely on the typical window painting mechanism for their drawing. Instead, Direct3D uses its own Render loop.</p> <p>Parameters:</p> <p>LPARAM (HDC) Handle to a drawing device context.</p>
<p><b>WM_KEYDOWN</b></p>	<p>Sent to an application whenever a user presses a key on the keyboard. Also sent repeatedly if a user holds down a key for a specific period of time.</p> <p>Parameters:</p> <p>LPARAM (int) Virtual key code for the pressed key.</p>
<p><b>WM_KEYUP</b></p>	<p>Sent to an application whenever a pressed key on the keyboard is released.</p> <p>Parameters:</p> <p>LPARAM (int) Virtual key code for the pressed key.</p>
<p><b>WM_CHAR</b></p>	<p>Sent to an application whenever a user presses a non-control key on the keyboard.</p> <p>Parameters:</p> <p>LPARAM (int) Virtual key code for the pressed key.</p>
<p><b>WM_MOUSEMOVE</b></p>	<p>Sent to an application whenever the cursor position changes. In other words, whenever the user moves the mouse.</p> <p>Parameters:</p> <p>LOWORD LPARAM (int) X position of the mouse cursor.</p> <p>HIWORD WPARAM (int) Y position of the mouse cursor.</p>

<b>WPARAM</b>	<p>Indicates whether special control characters were pressed on the keyboard. This value can be a combination of one or more of the following constants.</p> <p><b>MK_CONTROL</b> — Control key was pressed.  <b>MK_LBUTTONDOWN</b> — Left mouse button is down.  <b>MK_MBUTTONDOWN</b> — Middle mouse button is down.  <b>MK_RBUTTONDOWN</b> — Right mouse button is down.  <b>MK_SHIFT</b> — Shift key is down.</p>
<b>WM_LBUTTONDOWN</b>	<p>Sent to an application whenever the left mouse button is pressed.</p> <p>Parameters:  Same as above.</p>
<b>WM_MBUTTONDOWN</b>	<p>Sent to an application whenever the middle mouse button is pressed.</p> <p>Parameters:  Same as above.</p>
<b>WM_RBUTTONDOWN</b>	<p>Sent to an application whenever the right mouse button is pressed.</p> <p>Parameters:  Same as above.</p>
<b>WM_LBUTTONUP</b>	<p>Sent to an application whenever the left mouse button is released.</p> <p>Parameters:  Same as above.</p>
<b>WM_MBUTTONUP</b>	<p>Sent to an application whenever the middle mouse button is released.</p> <p>Parameters:  Same as above.</p>
<b>WM_RBUTTONUP</b>	<p>Sent to an application whenever the right mouse button is released.</p> <p>Parameters:  Same as above.</p>

## 8.1.1 Posting Messages

Once messages are received by `WndProc`, they must be forwarded on to controls. More specifically, they must be forwarded to the topmost control of a hierarchy. Such a control might be an application's main window. The process of sending messages to controls is known as *dispatching*, and a function named `PostMessage` should be added to class `CXControl` for receiving these messages. Its declaration and definition will be examined later. For now it is sufficient to know that once a message arrives here, it must then be dispatched downward to its generations of children, a topic discussed in the next subsection. See the following `WndProc` function to see how selected messages are forwarded on to a control hierarchy.

```
LRESULT WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    switch (message)
    {
        case WM_MOUSEMOVE:
        case WM_KEYDOWN:
            //More messages here, you get the idea...
            Window->PostMessage(message, wParam, lParam, NULL);
            break;
    }
    return 0;
}
```

## 8.1.2 Message Specifics

Messages are posted to the topmost control in a hierarchy using the `PostMessage` method, as above. They are then passed down to its generations of children in the form of events. The nature of exactly how they are passed down and who receives them depends largely on the message; this is explored in subsequent sections.

## 8.2 Handling Mouse Messages

---

The topmost control in a hierarchy should commonly receive the following messages when a mouse state changes.

```
WM_LBUTTONDOWNBLCLK  
WM_LBUTTONDOWN  
WM_LBUTTONUP  
WM_MBUTTONDOWNBLCLK  
WM_MBUTTONDOWN  
WM_MBUTTONUP  
WM_MOUSEMOVE  
WM_RBUTTONDOWNBLCLK  
WM_RBUTTONDOWN  
WM_RBUTTONUP
```

Note that the list is not complete and most applications do not process middle mouse buttons. However, depending on the message, this could mean a button press, a button release, or a change of cursor position. The following subsections examine how to validate these mouse messages, and specifically how to pass them down to a child control in the form of an event. The following class declaration is of CXControl and includes additions for mouse events that will be developed throughout this section, the addition of the PostMessage message method, and several other functions that will be discussed shortly.

```
class CXControl  
{  
protected:  
    D3DXVECTOR2 m_Position;  
    DWORD m_Width;  
    DWORD m_Height;  
    bool m_Visible;  
    CTexture* m_Canvas;  
    CXPen* m_Pen;  
    CXControl* m_ChildControls;  
    CXControl* m_NextSibling;
```

```

CXControl* m_PreviousSibling;
CXControl* m_Parent;
public:
//Accessors
CXTTexture* GetCanvas(void) {return m_Canvas;}
void SetCanvas(CXTTexture* Texture) {m_Canvas = Texture;}
bool GetVisible(void) const {return m_Visible;}
void SetVisible(bool Visible) {m_Visible = Visible;}
CXPen* GetPen(void) const {return m_Pen;}
void SetPen(CXPen* Pen) {m_Pen = Pen;}
DWORD GetWidth(void) const {return m_Width;}
DWORD GetHeight(void) const {return m_Height;}
void SetWidth(DWORD Width) {m_Width = Width;}
void SetHeight(DWORD Height) {m_Height = Height;}
CXControl* GetParentControl(void) const {return m_Parent;}
void SetParentControl(CXControl* Control) {m_Parent = Control;}
CXControl* GetNextSibling(void) const {return m_NextSibling;}
void SetNextSibling(CXControl* Control) {m_NextSibling = Control;}
CXControl* GetPreviousSibling(void) const {return m_PreviousSibling;}
void SetPreviousSibling(CXControl* Control) {m_PreviousSibling = Control;}
CXControl* GetFirstChild(void) const {return m_ChildControls;}
void SetFirstChild(CXControl* Control) {m_ChildControls = Control;}
D3DXVECTOR2* GetPosition(void) const {return &m_Position;}
FLOAT GetXPos(void) const {return m_Position.x;}
FLOAT GetYPos(void) const {return m_Position.y;}
void SetXPos(FLOAT X) {m_Position.x = X;}
void SetYPos(FLOAT Y) {m_Position.y = Y;}
void SetXYPos(FLOAT X, FLOAT Y);
CXControl* AddChildControl(CXControl* Control);
CXControl* RemoveChildControl(CXControl* Control);
void RemoveAllChildren();
int GetChildCount();
void GetAbsolutePosition(D3DXVECTOR2* Position);

virtual void OnMouseDown(int Button, int X, int Y) = NULL;
//Called when the user moves the mouse cursor
virtual void OnMouseMove(int X, int Y) = NULL;
//Called when the user releases a mouse button
virtual void OnMouseUp(int Button, int X, int Y) = NULL;
//Called when a control should be drawn
bool PostMessage(UINT msg, WPARAM wParam, LPARAM lParam, void* Data);

```

```
//Determines whether a mouse cursor intersects the control
bool CursorIntersect(FLOAT X, FLOAT Y);
//Posts messages to all child controls

CXControl* PostToAll(UINT msg, WPARAM wParam, LPARAM lParam, void* Data);
};
```

## 8.2.1 Cursor Intersection

Once a mouse message is received by the `PostMessage` method, it is important to validate it. Because of the geometric nature of mouse actions, mouse clicks and movements will only affect the control beneath the cursor and not others beyond its range. Hence, only one control will eventually be notified of mouse messages by way of an event. To confirm that a control may potentially receive a mouse event, you must check whether the cursor position falls inside the rectangle of a control's canvas. The following `CursorIntersect` function has been added to `CXControl` and determines whether an input position matches this criterion.

```
bool CXControl::CursorIntersect(FLOAT X, FLOAT Y)
{
    D3DXVECTOR2 ControlAbsolutePos;
    ControlAbsolutePos.x = 0;
    ControlAbsolutePos.y = 0;

    GetAbsolutePosition(&ControlAbsolutePos);

    if((X >= ControlAbsolutePos.x) && (X <= ControlAbsolutePos.x +
        GetWidth()))
    if((Y >= ControlAbsolutePos.y) && (Y <= ControlAbsolutePos.y +
        GetHeight()))
        return true;

    return false;
}
```



**NOTE**

In Chapter 11, the mouse position is interpreted differently to handle the caret position in a text box control.

## 8.2.2 Hierarchical Posting

It has been mentioned that only one control at any one time can be affected by the mouse. In other words, only one control can reside directly beneath the cursor at once. Hence, this is the control that should receive a mouse event. Consequently, it is termed the *target control*. In this case, the target control of a hierarchy is the one whose canvas intersects the cursor and who has either no descendants or no descendants that also intersect the cursor. Effectively, the target is the lowest control in a hierarchy that intersects the cursor. Thus, determining this control is a process of elimination. This process begins at the top of a hierarchy and moves downward recursively until the target is found by intersection testing. Figures 8.2 through 8.5 demonstrate how this process operates diagrammatically.

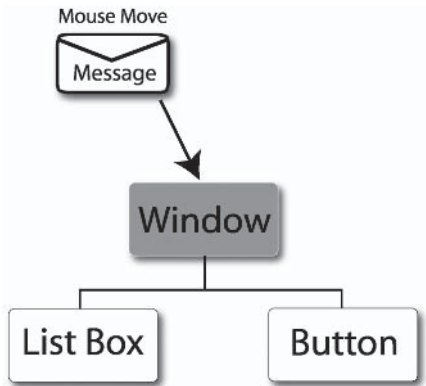


Figure 8.2. The application window receives a mouse move message.

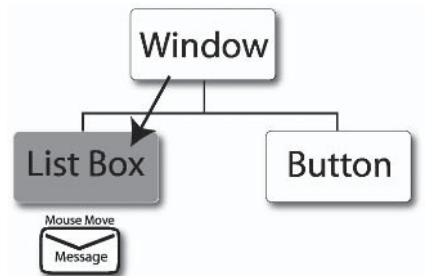


Figure 8.3. The message is passed sequentially to controls and their children.

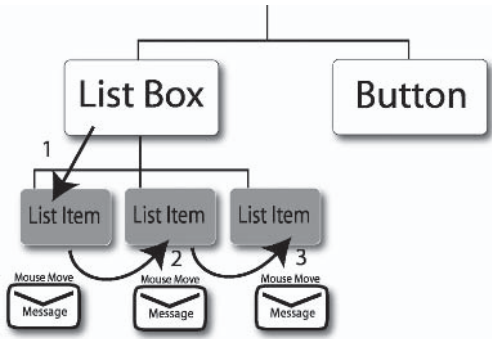


Figure 8.4. The message is passed along another generation sequentially.

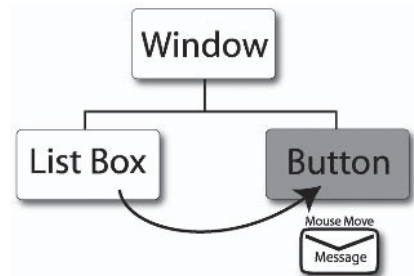


Figure 8.5. The message completes a generation and returns to the previous. Here, the process continues.

Correspondingly, a generic method called `PostToAll` has been added to `CXControl` (see below). It is a recursive function that posts messages to child controls down a hierarchy. The next subsections explore how this can be applied to handle mouse messages.

```

CXControl* CXControl::PostToAll(UINT msg, WPARAM wParam, LPARAM lParam,
                                void* Data)
{
    CXControl* Temp = GetFirstChild();

    while(Temp)
    {
        CXControl* Next = Temp->GetNextSibling();
        if(Temp->PostMessage(msg, wParam, lParam, Data))
            return Temp;

        Temp = Next;
    }

    return NULL;
}

```

### 8.2.3 Triggering Mouse Events

Here's where our hard work pays off, where we finally cut to the chase and turn mouse messages into mouse events for controls of a hierarchy. Previous sections explained how messages are posted from `WndProc` to the topmost control of a hierarchy using the `PostMessage` method of `CXControl`. They also explained how this function is ultimately responsible for triggering appropriate events in descendant classes. Consequently, this function can now begin to be implemented by handling mouse messages. The code below demonstrates how mouse messages received by `PostMessage` can be validated and dispatched to the target control.

```
bool CXControl::PostMessage(UINT msg, WPARAM wParam, LPARAM lParam,
                           void* Data)
{
    switch(msg)
    {
    case WM_LBUTTONDOWN:
        if(CursorIntersect(LOWORD(lParam), HIWORD(lParam)))
        {
            CXControl* Control = PostToAll(msg, wParam, lParam, Data);
            if(!Control)
                OnMouseDown(msg, LOWORD(lParam), HIWORD(lParam));
            return true;
        }
        else
            return false;
        break;
    }
}
```

## 8.3 Handling Keyboard Messages

---

The following keyboard messages occur whenever a user interacts with the keyboard:

```
WM_KEYDOWN
WM_KEYUP
WM_CHAR
```

A key can either be pressed or not pressed. As a result, keyboard messages are simpler to handle than mouse messages. However, keystrokes do not typically provide geometrically measurable data. Keys don't generally define a position on-screen like a cursor. Consequently, they cannot tell a program exactly which control input is intended for. This is why the concept of focus should be applied to handle keyboards. Basically, controls that are in focus receive keyboard messages. Furthermore, only one control in a hierarchy can be in focus at any time. Focus is examined in greater detail in the following subsection. The code below shows only the amended declaration for CXControl, the additions for implementing focus.

```
protected:
bool m_Focus;
//Focus control
CXControl* m_Focus;

public:
//Called when the user presses a key
virtual void OnKeyDown(WPARAM Key, LPARAM Extended) = NULL;
//Called when the user releases a key
virtual void OnKeyUp(WPARAM Key, LPARAM Extended) = NULL;
//Get focus control
CXControl* GetFocus() {return m_Focus;}
//Set focus control
void SetFocus(CXControl* Control);
};
```

### 8.3.1 Focus

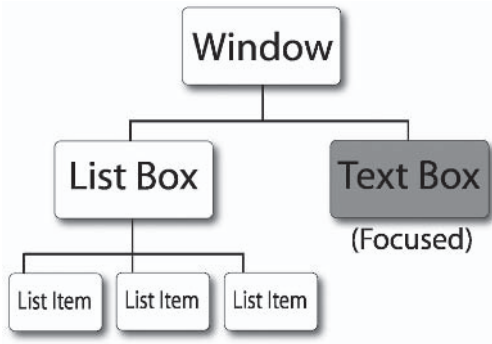


Figure 8.6

Controls in focus receive keyboard messages. Only one control of a hierarchy can be in focus at any given time. Focus is transferred to a control by selecting it; in other words, by clicking on it or through some other similar process. Essentially, as a control is brought into focus, another is brought out of focus. The following method of CXControl demonstrates how the singularity of focus is maintained throughout a hierarchy and how this can be transferred from one control to another.

```

void CXControl::SetFocus(CXControl* Control)
{
    if(!m_Focus)
    {
        if(GetParentControl())
            GetParentControl()->SetFocus(Control);
        else
        {
            if(GetFocus())
                GetFocus()->m_Focus= false;
            m_FocusControl = Control;
            m_Focus = true;
        }
    }
}
    
```

**NOTE**

You don't actually need to notify the parent control when its children gain focus. However, it has been implemented to do so here. This is so the focused control may directly receive input messages as they are posted to the topmost control in a hierarchy, as we will see in the next subsection.

## 8.3.2 Triggering Events

Turning keyboard messages into keyboard events is a simple process using the concept of focus. You simply invoke keyboard events in the focused control. The `PostMessage` method has been appended and now incorporates code to handle keyboard messages.

```
CXControl* CXControl::PostToAll(UINT msg, WPARAM wParam, LPARAM lParam,
                                void* Data)
{
    switch(msg)
    {
        //Handle other messages here...

        case WM_KEYUP:
        case WM_KEYDOWN:

            if(GetFocus())
            {
                if(msg == WM_KEYUP)
                    GetFocus()->OnKeyUp(wParam, lParam);

                if(msg == WM_KEYDOWN)
                    GetFocus()->OnKeyDown(wParam, lParam);
            }

            break;
    }
    return NULL;
}
```

## 8.4 Handling Control Painting

One of the final events to consider is also one of the most important — painting. Every control should receive an OnRender event whenever a hierarchy receives a WM\_RENDER message. This is a custom message that has been designated to substitute WM\_PAINT. It looks like this:

```
#define WM_RENDER WM_USER +1
```

This message indicates that a control is expected to redraw itself, a process known as *painting*. Each control paints itself differently; buttons have one look and labels have another, etc. The exact nature of how specific controls draw themselves is discussed as we consider different derived controls during subsequent chapters of Part II. Meanwhile, the following code is a sample OnRender event from CXControl. It demonstrates how a typical control might paint itself using its CXTexture and CXPen classes. The next section examines how paint messages are posted to controls through the hierarchy.

```
bool CXTest::OnRender()  
{  
    D3DXVECTOR2 ControlAbsolutePos;  
    ControlAbsolutePos.x = 0;  
    ControlAbsolutePos.y = 0;  
    GetAbsolutePosition(&ControlAbsolutePos);  
    GetCanvas()->SetTranslation(&ControlAbsolutePos);  
    GetPen()->DrawTexture(GetCanvas());  
    GetCanvas()->SetTranslation(NULL);  
}
```

**TIP**

WndProc receives `WM_PAINT` messages whenever the window should be repainted; however, Direct3D repaints windows during its own render procedure because the standard Windows painting simply isn't fast enough. Thus, `WM_PAINT` has been substituted by a custom-defined `WM_RENDER` message, and these are posted manually to a control hierarchy in a Direct3D application's render loop. You could just post a `WM_PAINT` message if you wanted, but `WM_RENDER` has been provided for clarity.

## 8.5 Posting in Reverse

---

Every control receives `OnRender` events as `WM_RENDER` is posted to the hierarchy. The painting process begins at the top of a hierarchy and moves downward progressively through generations. This makes sense because parent controls should be painted to the screen before their children. In other words, children shall be drawn after — on top of — their parents. However, the sequence in which children receive this event is important: They should be painted according to their *Z*-order. Specifically, this refers to the order in which controls are arranged in the third dimension. Those with lower *Z*-orders appear closer to the front of the screen, and those with higher *Z*-orders appear farther to the back, behind other controls. See Figure 8.7. Hierarchically, controls and their *Z*-orders look like Figure 8.8.

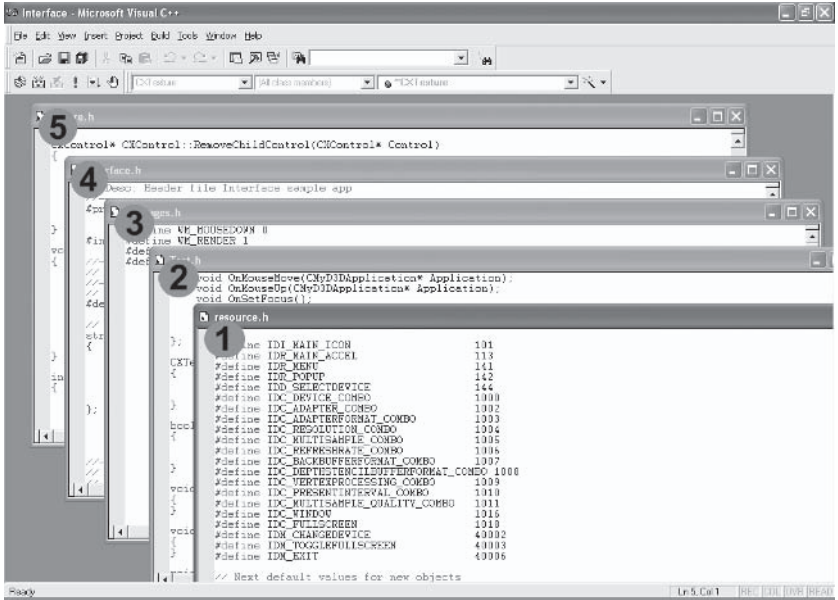


Figure 8.7. The Z-order of controls

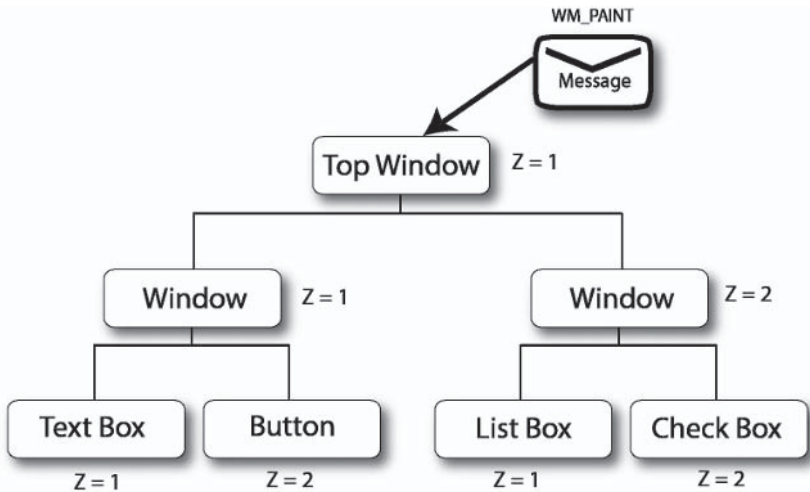


Figure 8.8. Hierarchy and Z-order of controls

As shown in Figure 8.8, siblings on the right have greater *Z* values than those to their left. This means that siblings on the right appear immediately behind those to their left. Thus, siblings of a hierarchy should be painted from right to left, not left to right. This ensures the leftmost control is painted at the front, on top of others. The code below is an adaptation of the `PostToAll` function, called `PostAllReverse`. Instead of passing a message downward to all controls from left to right, this posts a message downward from right to left.

```
CXControl* CXControl::PostToAllReverse(CXControl* Control, UINT msg,
                                       WPARAM wParam, LPARAM lParam, void* Data)
{
    CXControl* Next = Control->GetNextSibling();

    if(Next)
        Next->PostToAllReverse(Next, msg, wParam, lParam, Data);

    Control->PostMessage(msg, wParam, lParam, Data);

    return NULL;
}
```

## 8.6 Depth Sorting

---

If your desktop contains three different windows, you can usually switch from one to the next by clicking inside of them. As you do so, the selected window comes to the front and others are sent to the back. Effectively, their *Z*-order is changed. This is illustrated in the following figures.

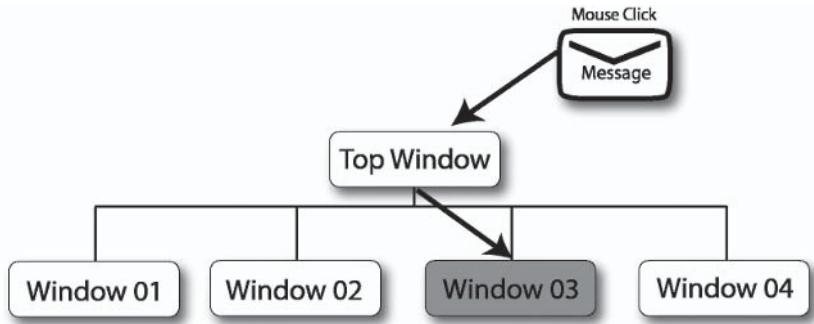


Figure 8.9. A window is selected by the mouse and must come to the front.

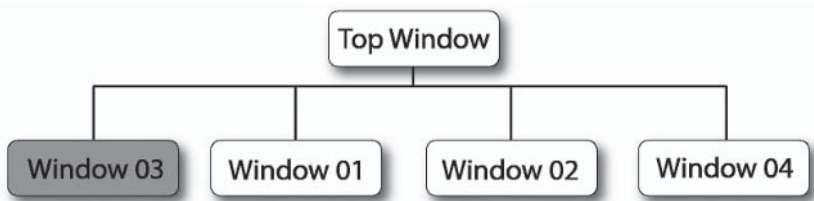


Figure 8.10. The window is moved to the left such that it has the lowest Z-order, and other windows also move along accordingly.

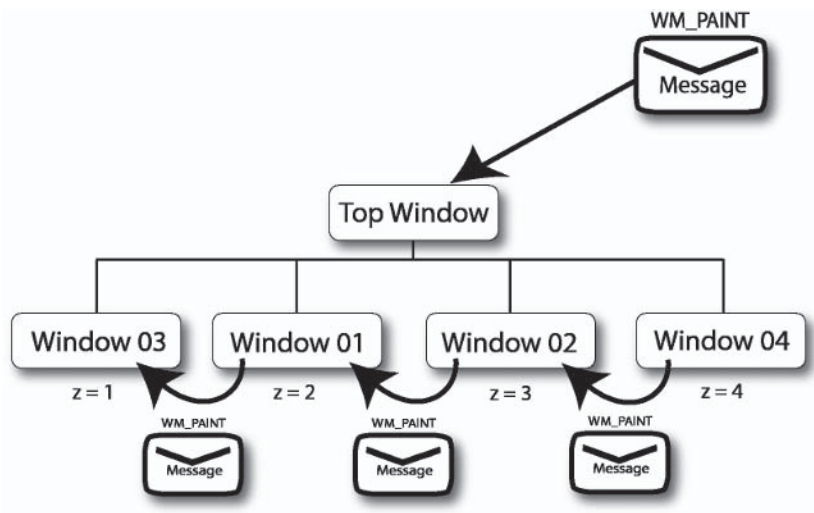


Figure 8.11. A paint message is received and windows are painted according to their Z-order.

To implement this in a control hierarchy, such that the currently activated control always remains on top of its siblings, you must ensure the activated control is also the leftmost control. If it isn't, then you must rearrange the controls so that it becomes so. The following function has been added to class `CXControl` and is named `MoveToFront`. It accepts a single control as an argument and makes it the leftmost node among its siblings.

```
void CXControl::MoveToFront(CXControl* Control)
{
    if(Control)
    {
        CXControl* Next = Control->GetNextSibling();           //Next
        CXControl* Previous = Control->GetPreviousSibling(); //Previous

        if(Previous) //Not the front
        {
            Previous->SetNextSibling(Next);

            if(Next)
                Next->SetPreviousSibling(Previous);
        }
        else
            return;

        Control->SetNextSibling(m_ChildControls);
        m_ChildControls->SetPreviousSibling(Control);
        Control->SetPreviousSibling(NULL);
        m_ChildControls = Control;
    }
}
```

## 8.7 Triggering Paint Events

Every control of a hierarchy effectively receives an OnRender event in reverse, from right to left. This is achieved by the PostAllReverse method and it ensures controls are drawn correctly to their Z-order. Thus, the PostMessage function can be amended to process WM\_RENDER messages as below. Notice the small amendments made to mouse click handling. It has been changed to set focus on a control, and has also been changed to use the MoveToFront method. This ensures controls become activated as they're clicked upon — in other words, moved to the leftmost node — and are then painted with the lowest Z-order.

```

CXControl* CXControl::PostToAll(UINT msg, WPARAM wParam, LPARAM lParam,
                                void* Data)
{
    switch(msg)
    {
        //Handle other messages here...
        case WM_LBUTTONDOWN:
        {
            if(CursorIntersect(LOWORD(lParam), HIWORD(lParam)))
            {
                CXControl* Control = PostToAll(msg, wParam, lParam, Data);
                if(!Control)
                {
                    OnMouseDown(LOWORD(lParam), HIWORD(lParam));
                    if(GetParentControl())
                        GetParentControl()->MoveToFront(this);

                    SetFocus(this);
                }
                return true;
            }
            break;
        }
    }
}
    
```

```
case WM_PAINT:
{
    if(GetVisible())
    {
        OnRender();

        if(m_ChildControls)
            PostToAllReverse(m_ChildControls, msg, wParam, lParam, Data);
    }
    break;
}
}

return NULL;
}
```

## 8.8 CXCControl — The Final Declaration

---

```
class CXCControl
{
protected:
    D3DXVECTOR2 m_Position;
    DWORD m_Width;
    DWORD m_Height;
    bool m_Visible;
    CXTTexture* m_Canvas;
    CXPen* m_Pen;
    CXCControl* m_ChildControls;
    CXCControl* m_NextSibling;
    CXCControl* m_PreviousSibling;
    CXCControl* m_Parent;
public:
    //Accessors
    CXTTexture* GetCanvas(void) const {return m_Canvas;}
    void SetCanvas(CXTTexture* Texture) {m_Canvas = Texture;}
    bool GetVisible(void) const {return m_Visible;}
    void SetVisible(bool Visible) {m_Visible = Visible;}
    CXPen* GetPen(void) const {return m_Pen;}
```

```

void SetPen(CXPen* Pen) {m_Pen = Pen;}
DWORD GetWidth(void) const {return m_Width;}
DWORD GetHeight(void) const {return m_Height;}
void SetWidth(DWORD Width) {m_Width = Width;}
void SetHeight(DWORD Height) {m_Height = Height;}
CXControl* GetParentControl(void) const {return m_Parent;}
void SetParentControl(CXControl* Control) {m_Parent = Control;}
CXControl* GetNextSibling(void) const {return m_NextSibling;}
void SetNextSibling(CXControl* Control) {m_NextSibling = Control;}
CXControl* GetPreviousSibling(void) const {return m_PreviousSibling;}
void SetPreviousSibling(CXControl* Control) {m_PreviousSibling = Control;}
CXControl* GetFirstChild(void) const {return m_ChildControls;}
void SetFirstChild(CXControl* Control) {m_ChildControls = Control;}
D3DXVECTOR2* GetPosition(void) const {return &m_Position;}
FLOAT GetXPos(void) const {return m_Position.x;}
FLOAT GetYPos(void) const {return m_Position.y;}
void SetXPos(FLOAT X) {m_Position.x = X;}
void SetYPos(FLOAT Y) {m_Position.y = Y;}
void SetXYPos(FLOAT X, FLOAT Y);
CXControl* AddChildControl(CXControl* Control);
CXControl* RemoveChildControl(CXControl* Control);
void RemoveAllChildren();
int GetChildCount();
void GetAbsolutePosition(D3DXVECTOR2* Position);
virtual void OnMouseDown(int Button, int X, int Y) = NULL;
virtual void OnMouseMove(int X, int Y) = NULL;
virtual void OnMouseUp(int Button, int X, int Y) = NULL;
virtual bool OnRender(void) = NULL;
bool PostMessage(UINT msg, WPARAM wParam, LPARAM lParam, void* Data);
bool CursorIntersect(FLOAT X, FLOAT Y);
CXControl* PostToAll(UINT msg, WPARAM wParam, LPARAM lParam, void* Data);

//Called when the user presses a key
virtual void OnKeyDown(WPARAM Key, LPARAM Extended) = NULL;
//Called when the user releases a key
virtual void OnKeyUp(WPARAM Key, LPARAM Extended) = NULL;
//Focus control
CXControl* m_Focus;
//Get focus control
CXControl* GetFocus() const {return m_Focus;}

```

```
//Set focus control  
void SetFocus(CXControl* Control);  
CXControl* PostToAllReverse(CXControl* Control, UINT msg, WPARAM wParam,  
                             LPARAM lParam, void* Data);  
  
};
```

## 8.9 Conclusion

---

This chapter completed the base class CXControl. This now contains everything needed to derive controls from it, a process that begins in the next chapter with CXWindow. Before moving on, however, let's review what this chapter has demonstrated.

- Messages are data structures that describe events on a user's computer. These can include paint messages, mouse messages, keyboard messages, etc.
- Messages are posted to the topmost control in a hierarchy. They should be filtered downward to child controls in the form of events.
- Mouse messages describe mouse related events. These involve button presses, button releases, and cursor movements.
- Only one control can be affected by the mouse at once, and this should receive notifications by way of an event. These take the form of OnMouseMove, OnButtonDown, and OnButtonUp.
- Keyboard messages describe keyboard related events. These involve keypresses and key releases.
- Because keyboards cannot define a position on-screen like a mouse cursor, they cannot typically identify which control should receive input. For this reason, the concept of focus is introduced.

- Controls that have focus receive keyboard input exclusively. Only one control in a hierarchy can be in focus at any one time.
- WM\_PAINT messages indicate that controls should repaint themselves.
- Controls should be drawn according to their Z-order. This defines where controls are positioned in relation to one another in the third dimension.

This page intentionally left blank.

# Developing Windows

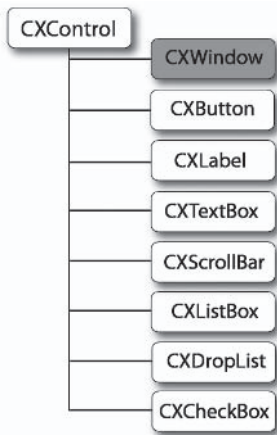


Figure 9.1

This chapter discusses the development of UI LIB's first independent control, `CXWindow`. This is derived from the base class `CXControl`, and represents a standard window. It has the potential to contain other controls and allows both dragging and minimizing. Overall, this chapter investigates the following issues:

- Definition of a window
- Deriving from `CXControl`
- Loading a window background

- Handling events
- Painting a window
- Implementing window dragging
- Minimizing and restoring
- Using `CXWindow` and `CXDesktop`

## 9.1 **CXWindow — Deriving from CXControl**

---

`CXWindow` will be a control. Like all controls in UI LIB, it derives from base class `CXControl`. This means that it inherits from `CXControl` as well as extends upon it. Chapters 7 and 8 demonstrated how `CXControl` is a model, or a blueprint, defining only the characteristics common to all controls. In effect, `CXControl` represents the minimum set of features that a control must have to be considered a control. Deriving controls from here therefore saves us having to individually define this feature set in each derived control. To see a list of the common features `CXWindow` will be inheriting, see Table 9.1.

**Table 9.1. `CXWindow` features**

Canvas	The rectangular drawing area upon which a control paints itself. Essentially, a canvas is the visible body of a control; it is how the user will see it.
Position	Controls have a position in terms of X and Y. This is usually a relative distance measured in pixels from the origin of their parent control. However, if a control is the topmost in a hierarchy and therefore has no parent, the relative and absolute position is the same since it defines a relative distance from the desktop's origin.
Size	Controls have a size in terms of width and height. Again, this is measured in pixels.
Events	Controls must respond to specific events, such as mouse clicks, keypresses, and screen refreshes.

## 9.2 Desktop and Application Windows

---

CXWindow is a single class that encapsulates a window. More specifically, it has the potential to encapsulate either of two window types: a parent window or a child window. These types are explained later. Any single instance of CXWindow can represent either form but not both. The class declaration for CXWindow appears below; notice the overridden events and addition of a constructor. Following sections add to this class and examine in more detail the two forms of windows it encapsulates.

```
class CXWindow : public CXControl
{
protected:
bool IsParentWindow; //Does this class represent a child or parent window?
public:
CXWindow(bool WindowType);
bool OnRender(WPARAM Key, LPARAM Extended);
void OnMouseDown(int Button, int X, int Y);
void OnMouseMove(int X, int Y);
void OnMouseUp(int Button, int X, int Y);
void OnSetFocus();
void OnLostFocus();
void OnKeyDown(WPARAM Key, LPARAM Extended);
void OnKeyUp(WPARAM Key, LPARAM Extended);
bool GetWindowType() const {return WindowType;}
void SetWindowType(bool Type) {WindowType = Type;}
};
```

### 9.3 Class CXWindow as a Parent

---

The first window style to consider is analogous to the desktop, the parent window. Generally, it is the top frame, or top window, in an application of which all other windows are children — much like an MDI program. Thus, our Direct3D applications only have one parent window. To UI LIB, this is hierarchically the topmost control

where all messages are posted and then dispatched downward through child controls. Practically, the parent window will be invisible and sized to match the client area of the application window. Figures 9.2 and 9.3 illustrate the parent window in terms of being like a desktop.

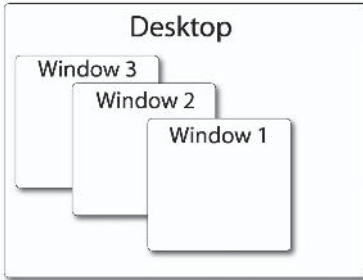


Figure 9.2



Figure 9.3

These are the most important features of the parent window:

- The parent window is the topmost control in a hierarchy. It has no parent itself, except for the screen.
- As a window, it cannot be dragged, moved, minimized, or resized. It must remain static and in one place, like a permanently fixed frame.
- Its canvas represents the greatest extents of an interface in terms of width and height. Every control exists within its boundaries and nothing can pass through them.
- It is essentially the representative of an interface. All messages and events should be posted here only, using the `PostMessage` method of `CXControl`. From here, the desktop will implicitly handle how messages are then dispatched to children in the form of events.
- Being the topmost control in a hierarchy, the parent window effectively controls the lifetime of every object in an interface. When the parent is removed from memory, all children will be too.

## 9.4 Implementing the Parent Window

Considering the power the parent has and its importance for an interface, `CXWindow` in terms of the parent window is actually relatively simple to implement. This is primarily because most of the parent's functionality has been already been implemented into `CXControl`. It can already become part of a hierarchy, already pass messages downward as events, and already has width and height. In fact, there is nothing further that *needs* to be added to this class to make it a suitable parent frame. Of course, you could add a function to load an image onto its canvas to use a kind of desktop background. However, image loading is something we will consider soon as we examine `CXWindow` in terms of a standard child window. This is something that requires more thought, and it is to this subject that we must now turn.

## 9.5 `CXWindow` as a Child Window

The second form of window that `CXWindow` encapsulates is a child window. This is more prevalent and in keeping with our understanding of a window. It is a Direct3D version of the typical frame that can be dragged and minimized, and applications can have as many instances of it as they want. It will be like all the child windows of an MDI application. Figures 9.4 and 9.5 illustrate a child window.

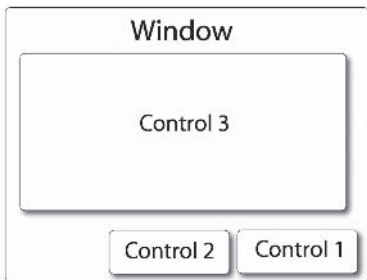


Figure 9.4

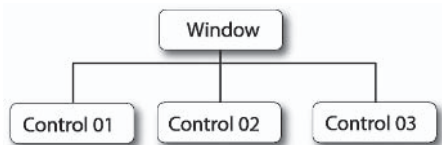


Figure 9.5

These are some of the most important properties of child windows:

- Child windows, as all controls, are children of the parent window or ultimately trace their ancestry to this control. In other words, child windows are descendants of the parent but *can* be children to other child windows too.
- Child windows define the greatest geometrical extents in terms of width and height beyond which its children cannot exist. In other words, controls can move freely within their window's boundaries but not beyond them. To this extent, windows are a container for controls.
- Unless invisible, child windows always paint upon their canvas. They might show images loaded from files or elsewhere. Ultimately, this image acts as a window background.
- Child windows can be dragged around the screen. Because its child controls are dependent upon the window for their position and size, as a window is dragged all contained controls should logically follow.
- Like the parent window, controls depend upon a child window for subsistence. Effectively, this means controls cannot exist if their parent is destroyed. If a window is removed from memory, its children will follow.

## 9.6 Implementing Child Windows

---

Child windows require more work to implement than parent windows. This is because we must code specific behaviors like painting, dragging, and minimizing. Consequently, the following functions and properties have been added to class `CXWindow`, and details of these features are discussed in following subsections.

```
class CXWindow : public CXControl
{
protected:
    bool IsParentWindow;
    bool m_IsMouseDown;
```

```

int m_LastNormalPosX;
int m_LastNormalPosY;
int m_LastNormalWidth;
int m_LastNormalHeight;
bool m_Minimized;
public:
CXWindow(bool WindowType);
bool OnRender(WPARAM Key, LPARAM Extended);
void OnMouseDown(int Button, int X, int Y);
void OnMouseMove(int X, int Y);
void OnMouseUp(int Button, int X, int Y);
void OnSetFocus();
void OnLostFocus();
void OnKeyDown(WPARAM Key, LPARAM Extended);
void OnKeyUp(WPARAM Key, LPARAM Extended);
bool GetWindowType() const {return WindowType;}
void SetWindowType(bool Type) {WindowType = Type;}
bool LoadCanvasFromFile(char* File);
void Minimize(int Width, int Height, int X, int Y);
void Restore(void);
};

```

### 9.6.1 Child Windows — Loading the Canvas

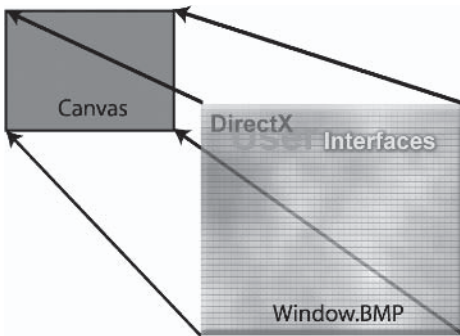


Figure 9.6

One of the first things an application will want to do with a child window is load an image onto its canvas to use as a background and to give the window a look rather than being transparent, as shown

in Figure 9.6. You'll recall from previous chapters that the canvas is declared as an instance of `CXTexture`, and this represents a collection of pixels. Thus, loading an image to the window's canvas is *almost* as simple as loading an image onto a texture. I say "almost" because you must also be sure to size the window's width and height according to the image's dimensions. If you don't do that, the window may not receive mouse messages because the cursor intersection will not be working with the correct dimensions. The image loading process can be coded into a method called `LoadCanvasFromFile`, and its definition is as follows.

```
bool CXWindow::LoadCanvasFromFile(char* File)
{

    if(GetWindowType() != WINDOW_TYPE_DESKTOP)
    {
        D3DXIMAGE_INFO Info;

        if(SUCCEEDED(D3DXGetImageInfoFromFile(File, &Info))
        {
            SetWidthHeight(Info.Width, Info.Height);
            if(SUCCEEDED(GetCanvas()->LoadFromFile(File)))
                return true;
        }
    }
    return false;
}
```

## 9.6.2 Painting Application Windows

The only way a window or any control displays itself on-screen is by painting its canvas. In other words, a control displays itself by sending its canvas pixels to the screen. This process should occur whenever the class receives an `OnRender` event, and it simply involves drawing an instance of `CXTexture` to the screen using `CXPen`. The `OnRender` event demonstrates how this is done.

```
bool CXWindow::OnRender()
{
    if(GetWindowType() != WINDOW_TYPE_DESKTOP)
    {
        D3DXVECTOR2 ControlAbsolutePos;

        ControlAbsolutePos.x = 0;
        ControlAbsolutePos.y = 0;

        GetAbsolutePosition(&ControlAbsolutePos);

        GetCanvas()->SetTranslation(&ControlAbsolutePos);

        Result = GetPen()->DrawTexture(GetCanvas());

        GetCanvas()->SetTranslation(NULL);
    }
    return true;
}
```

### 9.6.3 Dragging Application Windows

To make your windows more like the real thing, you're going to want to be able to drag them. In other words, if the user clicks and holds the left mouse button inside the window — like the title bar — you'll want it to follow the path of the cursor as long as the mouse button is held down. See Figure 9.7.

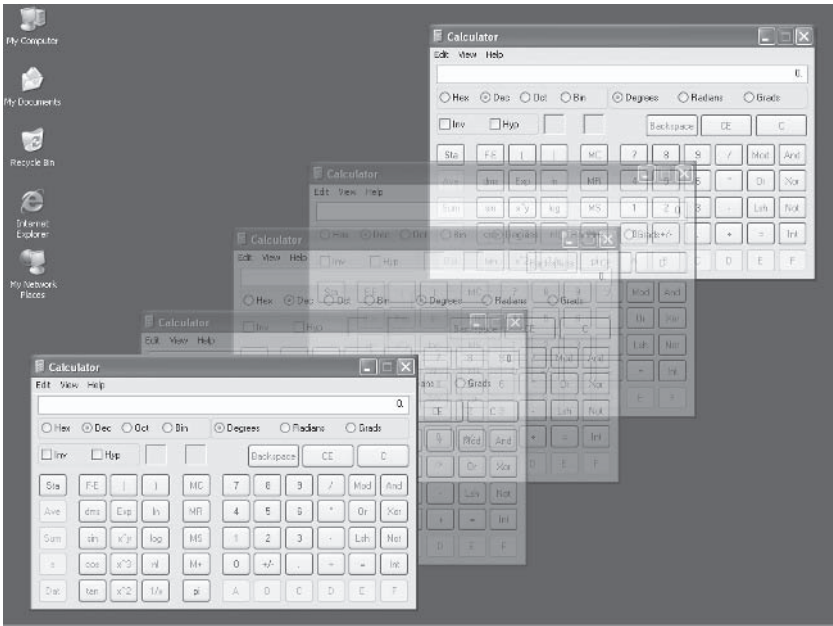


Figure 9.7

Implementing this functionality can sometimes be a confusing thing to figure out in your mind even though it requires only a few lines of code. Essentially, you should perform the following steps to achieve this. Take a look at the code for the `MouseDown`, `MouseMove`, and `MouseUp` events to see how window dragging is implemented in `CXWindow`.

1. Maintain a Boolean class member (`m_IsButtonDown`) to indicate whether the mouse button is currently pressed or not. This will be set accordingly on both `MouseDown` and `MouseUp` events.
2. On an `OnMouseDown` event, record the relative (X,Y) distance of the mouse cursor from the origin of the window. This will be used later as an offset to compute the new position of the window as the mouse cursor moves.

- Then, during OnMouseMove events, use the following formula to compute the new position of a window:

$$\text{New Window Pos} = \text{CurrentXPos} + (\text{Mouse X Pos} - \text{CurrentXPos}) - \text{Mouse Click Pos}$$

```

void CXTest2::OnMouseDown(int Button, int X, int Y)
{
    if(GetWindowType() != WINDOW_TYPE_DESKTOP)
    {
        D3DXVECTOR2 Abs;
        Abs.x = 0;
        Abs.y = 0;
        GetAbsolutePosition(&Abs);

        m_X = X - Abs.x;
        m_Y = Y - Abs.y;
        m_Depressed = false;
    }
}

void CXTest2::OnMouseMove(int X, int Y)
{
    if(GetWindowType() != WINDOW_TYPE_DESKTOP)
    {
        if(!m_Depressed)
        {
            D3DXVECTOR2 Abs;
            Abs.x = 0;
            Abs.y = 0;
            GetAbsolutePosition(&Abs);

            SetXPos(GetXPos() + ((X - Abs.x) - m_X));
            SetYPos(GetYPos() + ((Y - Abs.y) - m_Y));
        }
    }
}
    
```

```
void CXTest2::OnMouseUp(int Button, int X, int Y)
{
    if(GetWindowType() != WINDOW_TYPE_DESKTOP)
        m_Depressed = true;
}
```

## 9.6.4 Minimizing and Restoring Application Windows

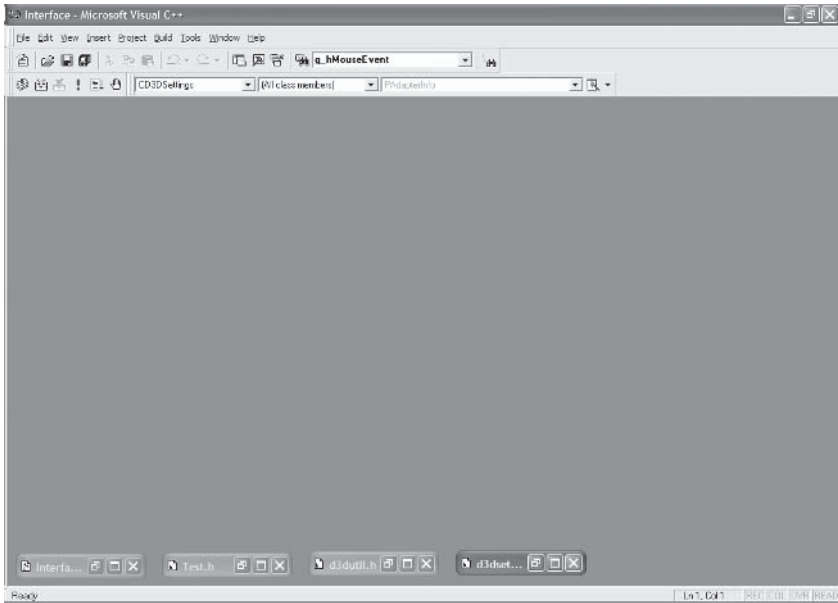


Figure 9.8. Minimized windows

Minimized windows are those which have been shrunk to their smallest extent at the bottom of the screen (see Figure 9.8). This usually occurs because the user doesn't need to use them immediately but plans on doing so later on. Restored windows, on the other hand, are those which a user has brought back to their original size and position after being minimized (see Figure 9.9).

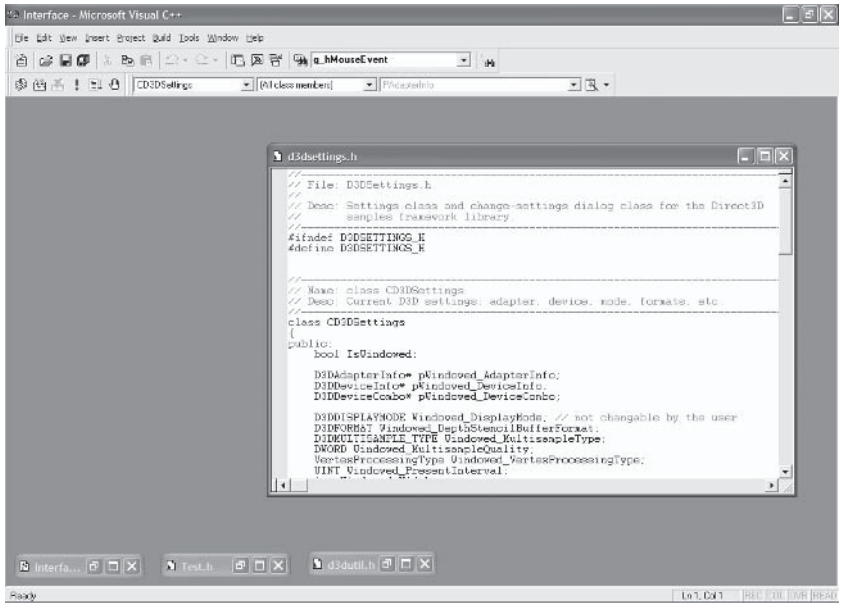


Figure 9.9. Restored window

Implementing similar behaviors into `CXWindow` should not be too difficult. Effectively, you will need to maintain two sizes and positions, one for the minimized state and one for the normal state, and you will also use a Boolean variable to indicate which state the window is in. To minimize a window, simply make all of its child controls invisible, shrink the window's width and height to a proportion of its original size or to a specified size, and reposition the window strategically at the bottom of the screen or another discreet but accessible location. You may also be inclined to perform some transitional animation to illustrate what has occurred. To restore the window, you simply switch the window state back to its original properties. Easy! The following `Minimize` and `Restore` methods of `CXWindow` demonstrate how a window can be minimized and restored.

```
void CXWindow::Minimize(int Width, int Height, int X, int Y)
{
    if(GetWindowType() != WINDOW_TYPE_DESKTOP)
    {
        CXControl* Temp = this->GetFirstChild();

        while(Temp)
        {
            CXControl* Next = Temp->GetNextSibling();
            Temp->SetVisible(false);
            Temp = Next;
        }

        m_LastNormalPosX = this->GetXPos();
        m_LastNormalPosY = this->GetYPos();
        m_LastNormalHeight = this->GetWidth();
        m_LastNormalWidth = this->GetHeight();
        m_Minimized = true;

        this->SetXYPos(X,Y);
        this->SetWidthHeight(Width, Height);
    }
}

//-----

void CXWindow::Restore(void)
{
    if(GetWindowType() != WINDOW_TYPE_DESKTOP)
    {
        this->SetXYPos(m_LastNormalPosX, m_LastNormalPosY);
        this->SetWidthHeight(m_LastNormalWidth, m_LastNormalHeight);
        m_Minimized = false;
    }
}
```

# 9.7 Using CXWindow — Sample Application

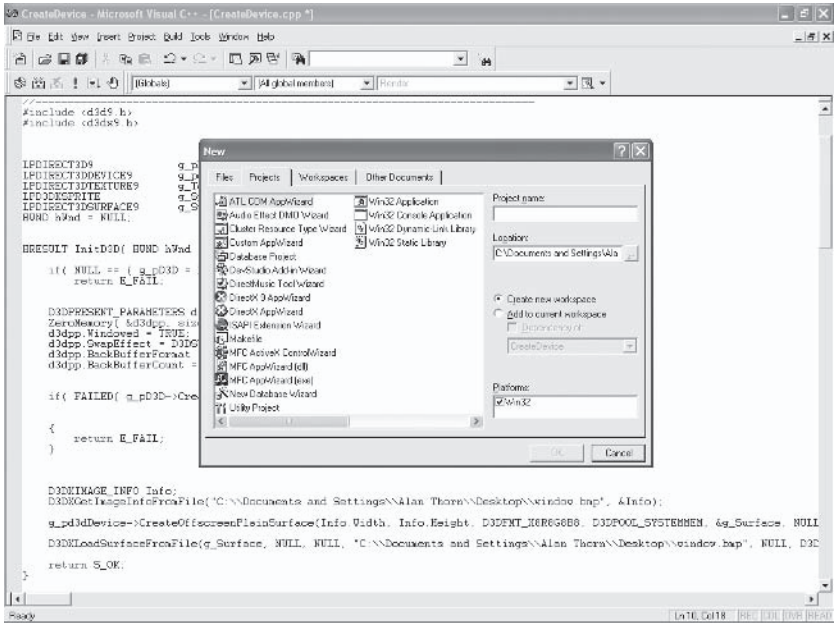


Figure 9.10

CXWindow is now complete. UILIB.H now contains something a developer can use. Finally, a point has been reached where the work over these past three chapters can be put to the test in the form of a sample application containing a single window. Of course, to contain one window, two instances of CXWindow must actually be used, one for the parent window — the container — and one for the window itself. Obviously, were you to add more windows you do not need to add more parents; you make all controls children of the same parent. You can either start a new Direct3D project in Microsoft Visual C++, as described in Chapter 3, or load this program from the book’s companion CD. The application is small and the complete code follows.

UI LIB

```

#include <d3d9.h>
#include <d3dx9.h>
#include <UILIB.H>

LPDIRECT3D9          g_pD3D          = NULL;
LPDIRECT3DDEVICE9   g_pd3dDevice = NULL;
CXWindow * g_Parent; //Main desktop window
CXWindow* g_Window; //A standard window

//-----
// InitD3D
//-----
HRESULT InitD3D(HWND hWnd)
{
    if(NULL == (g_pD3D = Direct3DCreate9(D3D_SDK_VERSION)))
        return E_FAIL;

    D3DPRESENT_PARAMETERS d3dpp;
    ZeroMemory(&d3dpp, sizeof(d3dpp));
    d3dpp.Windowed = TRUE;
    d3dpp.SwapEffect = D3DSWAPEFFECT_DISCARD;
    d3dpp.BackBufferFormat = D3DFMT_UNKNOWN;

    if(FAILED(g_pD3D->CreateDevice(D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL, hWnd,
        D3DCREATE_SOFTWARE_VERTEXPROCESSING,
        &d3dpp, &g_pd3dDevice)))
    {
        return E_FAIL;
    }

    g_Parent = new CXWindow(WINDOW_TYPE_DESKTOP); //Create desktop
                                                //window
    g_Parent ->SetWidthHeight(640,480); //Make desktop match window size
    g_Parent ->SetXYPos(0,0); //Set desktop to window origin

    //Create standard window
    g_Window = new CXWindow(WINDOW_TYPE_STANDARD);
    g_Window->LoadCanvasFromFile("test.jpg");
    g_Window->SetXYPos(0,0);

```

```

g_Parent ->AddChildControl(Window); //Set standard window as child
                                     //of desktop

    return S_OK;
}

//-----
// Cleanup
//-----
VOID Cleanup()
{

    delete g_Parent; //Delete desktop and all children

    if(g_pd3dDevice != NULL)
        g_pd3dDevice->Release();

    if(g_pD3D != NULL)
        g_pD3D->Release();
}

//-----
// Render
//-----
VOID Render()
{
    if(NULL == g_pd3dDevice)
        return;

    g_pd3dDevice->Clear(0, NULL, D3DCLEAR_TARGET, D3DCOLOR_XRGB(0,0,255),
        1.0f, 0);

    if(SUCCEEDED(g_pd3dDevice->BeginScene()))
    {
        g_Parent ->PostMessage(WM_RENDER, ,0,0, NULL); //Post WM_PAINT
                                                    //message

        g_pd3dDevice->EndScene();
        g_pd3dDevice->Present(NULL, NULL, NULL, NULL);
    }
}

```

```

}

//-----
// MsgProc
//-----
LRESULT WINAPI MsgProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
    switch(msg)
    {
        case WM_DESTROY:
            Cleanup();
            PostQuitMessage(0);
            return 0;

        case WM_PAINT:
            ValidateRect(hWnd, NULL);
            return 0;

        case WM_MOUSEMOVE:
        case WM_LBUTTONDOWN:
        case WM_LBUTTONUP:
        case WM_KEYUP:
        case WM_KEYDOWN:
            //Post other messages
            g_Desktop->PostMessage(msg, wParam, lParam, NULL);
            return 0;
    }

    return DefWindowProc(hWnd, msg, wParam, lParam);
}

//-----
// WinMain – Application starts here
//-----
INT WINAPI WinMain(HINSTANCE hInst, HINSTANCE, LPSTR, INT)
{

```

```

WNDCLASSEX wc = {sizeof(WNDCLASSEX), CS_CLASSDC, MsgProc, 0L, 0L,
    GetModuleHandle(NULL), NULL, NULL, NULL, NULL, "D3D", NULL};
RegisterClassEx(&wc);

HWND hWnd = CreateWindow("D3D", "Windows", WS_OVERLAPPEDWINDOW, 100,
    100, 1024, 768, GetDesktopWindow(), NULL,
    wc.hInstance, NULL);

if(SUCCEEDED(InitD3D(hWnd)))
{
    ShowWindow(hWnd, SW_SHOWDEFAULT);
    UpdateWindow(hWnd);

    MSG msg;

    PeekMessage(&msg, NULL, 0, 0, PM_NOREMOVE);

    while(msg.message != WM_QUIT)
    {
        if (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
        {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
        else
            Render();
    }
}

UnregisterClass("D3D", wc.hInstance);
return 0;
}

```

## 9.7.1 Overview

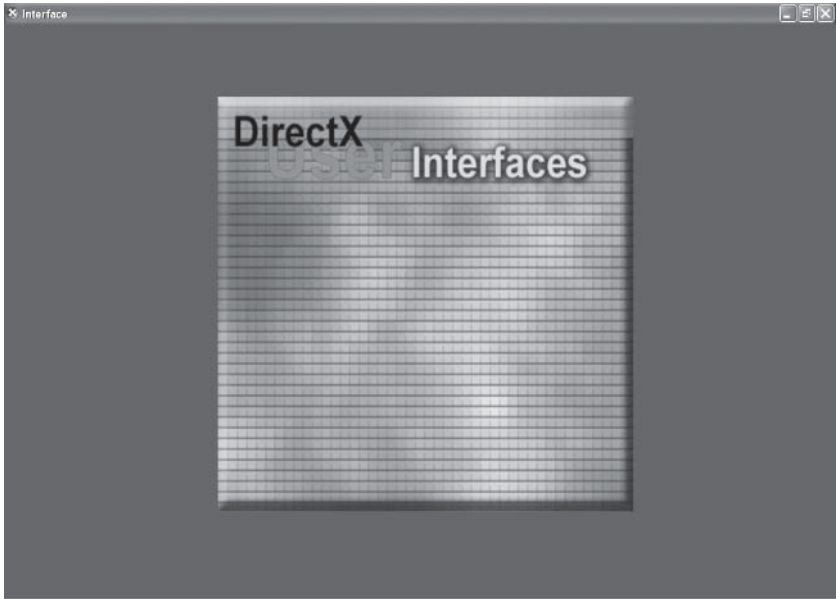


Figure 9.11

This sample application displays a window that can be dragged and minimized and which loads its canvas from an image file. It is declared as `g_Window` and is a child of the parent window, which is declared as `g_Parent`. The following sections examine specific areas in more detail.

## 9.7.2 Desktop Initialization

Initialization of most interfaces occurs with the topmost control of a hierarchy. In this case, it is the parent, which is like the desktop. The result of this instantiation immediately leaves us with a hierarchy of one control, a control that has no width and no height and is positioned nowhere. These properties should then be set, as indeed they are. This occurs inside the application's `InitD3D` function, of which the relevant extract follows.

```
g_Parent = new CXWindow(WINDOW_TYPE_DESKTOP); //Create desktop window
g_Parent ->SetWidthHeight(640,480); //Make desktop match window size
g_Parent ->SetXYPos(0,0); //Set desktop to window origin
```

### 9.7.3 Window Initialization

Once a desktop has been created and a hierarchy is in place, controls can be added. This application shows only one window, represented by the pointer `g_Window`. Like the parent, this too is initialized inside the `InitD3D` procedure. This involves setting the window's position and background image and then adding it to the control hierarchy as a child of the desktop. The associated code can be seen below.

```
g_Window = new CXWindow(WINDOW_TYPE_STANDARD);
g_Window->LoadCanvasFromFile("test.jpg");
g_Window->SetXYPos(0,0);

g_Parent ->AddChildControl(Window); //Set standard window as child
//of desktop
```

### 9.7.4 Windows Message Posting

Messages are posted to the topmost control in a hierarchy, the desktop. These are then processed by `CXControl`'s `PostMessage` routine and are passed down the hierarchy in the form of events. You'll notice that messages are predominantly sent from an application's `WndProc` procedure, except for render messages. These are passed to controls from the rendering loop. See the following code.

```
VOID Render()
{
    if(NULL == g_pd3dDevice)
        return;
```

```
g_pd3dDevice->Clear(0, NULL, D3DCLEAR_TARGET, D3DCOLOR_XRGB(0,0,255),
                  1.0f, 0);

if(SUCCEEDED(g_pd3dDevice->BeginScene()))
{
    g_Parent ->PostMessage(WM_PAINT,0,0, NULL); //Post WM_PAINT message
    g_pd3dDevice->EndScene();
    g_pd3dDevice->Present(NULL, NULL, NULL, NULL);
}

}
```

## 9.7.5 Deleting an Interface

To some, it might seem surprising that `g_Window` is not explicitly released or deleted as the application ends. This is because deallocation of `g_Window` is done for us hierarchically as `g_Parent` is released. The parent control always controls the lifetimes of its children. Thus, as it is deleted, so are all of its children. The cleanup function is therefore very simple.

```
VOID Cleanup()
{

    delete g_Parent; //Delete desktop and all children

    if(g_pd3dDevice != NULL)
        g_pd3dDevice->Release();

    if(g_pD3D != NULL)
        g_pD3D->Release();

}
```

## 9.8 Conclusion

---

This chapter is probably one of the more satisfying for the developer because one can begin to see the rewards of his labors. CXWindow is not a class without the potential for expansion and improvement however. Some people with more time could go on to create resizable windows, dockable windows, and other clever effects. Overall, this chapter has explained the following:

- CXWindow derives from base class CXControl. CXControl defines a common feature set that is shared by all controls.
- CXWindow can encapsulate either of two forms of windows — a parent window or a child window.
- The parent window is the topmost control in a hierarchy. It represents the lifetime of an interface.
- Child windows are the typical frames that contain other controls and can be dragged and minimized.

This page intentionally left blank.

# Labels and Buttons

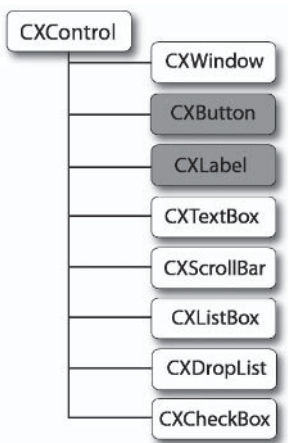


Figure 10.1

This chapter takes UI LIB yet another stage closer to completion by developing two further controls — labels and buttons. The former control displays text at a specified location and the latter performs operations when clicked. Overall, this chapter answers the following questions:

- What is a label?
- What is a button?
- What is the ID3DXFont interface?

## 10.1 Labels and Buttons

---

Both labels and buttons are designed to be children of a window control. Label controls are encapsulated into class `CXLabel` and button controls into class `CXButton`. Both of these are derived from `CXControl` and therefore inherit a control's common feature set, features that all controls must have to be considered a control. These have been discussed in previous chapters and can be said to include geometric properties like `Canvas`, `Position`, `Width`, and `Height`, and to support functional attributes like events. Subsequent sections in this chapter individually examine the label and button classes, beginning with labels.

## 10.2 CXLabel — Labels

---

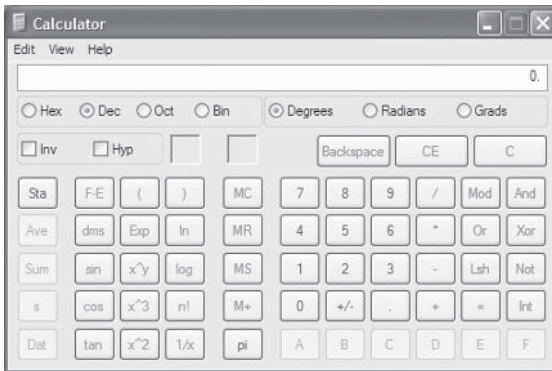


Figure 10.2

Labels are rectangular regions inside which text is displayed. Typically, interface developers employ them to achieve the tasks listed below.

- Labels are ideal for conveying on-screen instructions to users to help them use your software. This might consist of something as simple as “Click next to proceed” or something more

complex, depending on an application. However, it is important to remember the interface design guidelines from Chapter 1 and ensure that labels are as concise as possible.

- Labels are good for titling specific areas inside a window. This is particularly useful for spatially dividing controls that represent unrelated fields of information. For example, a customer details screen could divide its personal information section from its credit history section by labeling areas appropriately.
- One of the most common tasks labels are chosen for is the titling of input controls. In other words, labels are placed either beside or above controls like text boxes to indicate the kind of information it expects to be input, such as name, address, telephone number, etc.
- Label controls are also used by other controls to suit their textual needs. One of the clearest examples of this is check box controls. These are simply a check box control and a label control juxtaposed together, side by side. Others controls that typically use labels for their internal captioning properties include buttons, list boxes, and drop-down lists.

## 10.3 Labels as ID3DXFont

---

Class `CXLabel` encapsulates a label control. Take a look at its class declaration on the following page. Notice the inclusion of the protected member `ID3DXFont`. This is the interface through which `Direct3D` presents text on-screen, an interface not altogether dissimilar from `ID3DXSprite`. It is a powerful interface capable of showing text at specified positions, scaling text to fit inside a rectangle, showing text in a particular font, and showing text in a specified color. The next few sections of this chapter discuss the implementation of class `CXLabel`, and specifically how text is drawn to the screen using `ID3DXFont`.

```
class CXLable : public CXControl
{
protected:
    char m_Caption[255];
    LPD3DXFONT m_Font;
    D3DCOLOR m_Color;
    DWORD m_Format;
public:
    CXLable(LOGFONT Font, LPDIRECT3DDEVICE9 Device);
    ~CXLable();
    bool OnRender();
    void OnMouseDown(int Button, int X, int Y);
    void OnMouseMove(int X, int Y);
    void OnMouseUp(int Button, int X, int Y);
    void OnSetFocus();
    void OnLostFocus();
    void OnKeyDown(WPARAM Key, LPARAM Extended);
    void OnKeyUp(WPARAM Key, LPARAM Extended);
    void SetCaption(char* Caption);
    char* GetCaption() const {return m_Caption;}
    D3DCOLOR GetColor() const {return m_Color;}
    void SetColor(D3DCOLOR Color) {m_Color = Color;}
    DWORD GetFormat() const {return m_Format;}
    void SetFormat(DWORD Format) {m_Format = Format;}
};
```

### 10.3.1 Instantiating ID3DXFont

CXLable instantiates ID3DXFont inside its constructor. This can be done by calling either D3DXCreateFont or D3DXCreateFont-Indirect. The former function creates a font object from an explicitly specified font while the latter creates a font object from a font matching a specific criterion. The declaration for these two functions and their respective parameters follow.

```
HRESULT WINAPI D3DXCreateFont(
    LPDIRECT3DDEVICE9 pDevice,
    UINT Height,
    UINT Width,
    UINT Weight,
    UINT MipLevels,
    BOOL Italic,
    DWORD CharSet,
    DWORD OutputPrecision,
    DWORD Quality,
    DWORD PitchAndFamily,
    LPCTSTR pFacename,
    LPD3DXFONT *ppFont
);
```

---

**LPDIRECT3DDEVICE9 pDevice:** Pointer to the Direct3D device used to render text. For CXLabel, this value is passed as argument Device to the constructor.

---

**UINT Height:** Height of the characters in logical units.

---

**UINT Width:** Width of the characters in logical units.

---

**UINT Weight:** Typeface weight, such as bold.

---

**UINT MipLevels:** Mipmap levels.

---

**BOOL Italic:** True for italic font; false otherwise.

---

**DWORD CharSet:** Character set of the font.

---

**DWORD OutputPrecision:** Use OUT\_TT\_ONLY\_PRECIS to always get a TrueType font.

---

**DWORD Quality:** Does not affect TrueType fonts.

---

**DWORD PitchAndFamily:** Pitch and family index.

---

**LPCTSTR pFacename:** String containing the typeface name.

---

**LPD3DXFONT \*ppFont:** Address into which a font object is returned.

---

```

HRESULT WINAPI D3DXCreateFontIndirect(

    LPDIRECT3DDEVICE9 pDevice,
    CONST D3DXFONT_DESC *pDesc,
    LPD3DXFONT *ppFont
);

```

---

**LPDIRECT3DDEVICE9 pDevice:** Pointer to the Direct3D device used to render the text. For CXLLabel, this value is passed as argument Device to the constructor.

---

**CONST D3DXFONT\_DESC \*pDesc:** Pointer to a D3DXFONT\_DESC structure, describing properties about the font to be created.

---

**LPD3DXFONT \*ppFont:** Address into which a font object is returned.

---

CXLLabel uses the D3DXCreateFont function to create an ID3DXFont object. This is based upon the font handle passed as a parameter to the class constructor. The constructor's definition is as follows.

```

CXLLabel::CXLLabel(LOGFONT Font, LPDIRECT3DDEVICE9 Device) : CXControl(Device)
{

    D3DXCreateFont(Device, Font.lfWidth, Font.lfHeight, Font.lfWeight, 1,
                  Font.lfItalic, Font.lfCharSet, Font.lfOutPrecision,
                  Font.lfQuality, Font.lfPitchAndFamily,
                  Font.lfFaceName, &m_Font);
    m_Caption = "";
    m_Color = D3DCOLOR_XRGB(0,0,0);
    m_Format = DT_LEFT;
}

```

### 10.3.2 Setting the Label Caption

Once an application creates an instance of CXLabel it should then aim to set the label’s caption. The caption refers to the actual text that CXLabel displays when painted. This is stored in member m\_Caption and can be set and retrieved by its appropriate accessor methods. SetCaption is defined below.

```
void CXLable::SetCaption(char* Caption)
{
    if (Caption && *Caption)
        strcpy(m_Caption, Caption);
    else
        m_Caption[0] = 0;
}
```

### 10.3.3 Painting with ID3DXFont

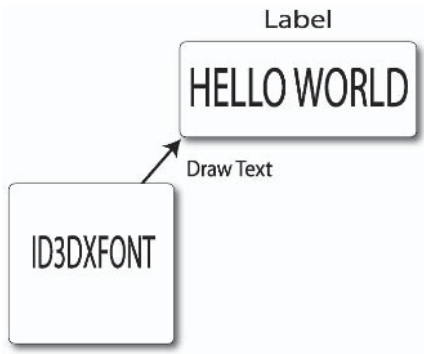


Figure 10.3

Controls paint themselves as OnRender events occur. For CXLable, this process involves painting the caption text inside its rectangular boundaries. This is achieved by calling the DrawText method of ID3DXFont. Its function definition and parameters follow.

```

INT DrawText(
    LPD3DXSPRITE pSprite,
    LPCTSTR pString,
    INT Count,
    LPRECT pRect,
    DWORD Format,
    D3DCOLOR Color
);

```

---

**LPD3DXSPRITE pSprite:** Just pass NULL for Direct3D to use a default. You could use your own sprite object to draw the text if you wish, however.

---

**LPCTSTR pString:** String to display, either ANSI or UNICODE. For CXLablel, this should equate to the caption property. If *Format* includes DT\_MODIFYSTRING, you should size your buffer to hold up to four additional characters.

---

**INT Count:** Indicates the length of *pString* in characters. To compute this value you could use the strlen function. Conversely, if *pString* is NULL terminated, you can pass -1.

---

**LPRECT pRect:** Pointer to a RECT structure that defines a rectangle into which the text should be drawn. For CXLablel, this region will represent its canvas. In terms of a RECT structure, this can be expressed as position, width, and height.

---

**DWORD Format:** Indicates how the text is presented. It can be one or more of the following values:

DT\_BOTTOM: Justifies text to the bottom of the rectangle. This must be combined with DT\_SINGLELINE.

DT\_CALCRECT: Automatically resizes *pRect* to fit the text by extending its width and height as required. Passing this parameter simply results in a new RECT, and no text is drawn. To draw text according to the new measurements, the function must be called again without passing this parameter.

DT\_CENTER: Centers text horizontally inside *pRect*.

DT\_EXPANDTABS: Expands tab characters, usually meaning a space of eight characters per tab.

DT\_LEFT: Text is aligned to the left.

**DT\_RIGHT:** Text is aligned to the right of *pRect*.

**DT\_RTLREADING:** Displays text from right to left when Hebrew or Arabic fonts are used.

**DT\_TOP:** Aligns text to the top of *pRect*.

**DT\_VCENTER:** Centers text vertically (single line only).

**DT\_WORDBREAK:** Breaks words; lines are broken between words where they extend past the edge of *pRect*.

**D3DCOLOR Color:** A D3DCOLOR structure defining which color the text should be. This could be specified by using the D3DCOLOR\_XRGB macro.

The OnRender event of CXLLabel can therefore be coded to paint its caption to the screen as follows.

```
bool CXLLabel::OnRender(void)
{
    if(m_Font)
    {
        RECT Rectangle;
        D3DXVECTOR2 Vec2;
        Vec2.x = 0;
        Vec2.y = 0;
        GetAbsolutePosition(&Vec2);
        Rectangle.left = Vec2.x;
        Rectangle.top = Vec2.y;
        Rectangle.right = GetWidth();
        Rectangle.bottom = GetHeight();
        m_Font->DrawText(NULL, m_Caption, strlen(m_Caption), &Rectangle,
                        m_Format, m_Color);
    }

    return true;
}
```



**NOTE**

In the next chapter we examine a more sophisticated way of storing and manipulating strings as we consider text boxes, specifically, `std:string`.

### 10.3.4 Releasing ID3DXFont

The last area of CXLablel to examine is the class destructor. Its definition probably comes as no surprise to anybody. It releases the ID3DXFont interface and clears the caption string. Take a look below.

```
CXLablel::~CXLablel()  
{  
    if(m_Font)  
        m_Font->Release();  
}
```

## 10.4 CXButton — Buttons

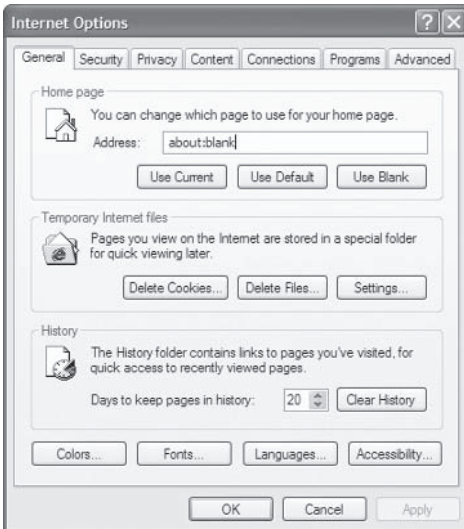


Figure 10.4

The final control to be considered in this chapter is one of the most prevalent, the button. The following are the key features of a button. These are developed in subsequent sections.

- A button control is a rectangular area and can be in one of two states: pressed or unpressed. This refers to whether the button has been clicked or not. You can determine this state through `OnMouseDown` and `OnMouseUp` events.
- Because a button can be in one of two states it will want to indicate which state it's currently in to a user. It usually does this by changing its appearance as it's clicked — *one* appearance for pressed states and *another* for unpressed states.
- Buttons typically have a visible caption, such as “OK” or “Cancel.” These are often used to describe the function of a button or the consequences that will arise from pressing it. These are implemented as a label.
- Buttons represent an action that occurs when clicked, like showing a window, deleting a file, or running a procedure. More commonly, this action is actually performed as a user releases the mouse button rather than presses it inside the boundary of a button.

## 10.5 CXButton — The Class Declaration

Button controls are encapsulated into class `CXButton`. It maintains a Boolean member to determine whether it is pressed or unpressed. It also has two textures corresponding to the pressed and unpressed images respectively, and maintains a label control for its caption property. The implementation details of this class are discussed in several of the following sections. Before jumping in, however, take a moment to examine the class declaration for `CXButton` below.

```
class CXButton : public CXControl
{
protected:
    HFONT m_Font;
    CXLable* m_Caption;
    CXTexture* m_DefaultImage;
```

```
    CTexture* m_PressedImage;
    bool m_Pressed;
public:
    CButton(LPDIRECT3DDEVICE9 Device);
    ~CButton();
    bool OnRender();
    void OnMouseDown(int Button, int X, int Y);
    void OnMouseMove(int X, int Y);
    void OnMouseUp(int Button, int X, int Y);
    void OnSetFocus();
    void OnLostFocus();
    void OnKeyDown(WPARAM Key, LPARAM Extended);
    void OnKeyUp(WPARAM Key, LPARAM Extended);
    void SetCaption(char* Caption);
    bool SetUnpressedImage(char* File);
    bool SetPressedImage(char* File);
};
```

## 10.5.1 The Class Constructor

CXButton uses its constructor to initialize many of its properties. These include two textures for its pressed and unpressed appearance, the pressed status Boolean, and its caption text. Its function definition is as below.

```
CXButton::CXButton(LPDIRECT3DDEVICE9 Device) : CXControl(Device)
{
    m_DefaultImage = new CTexture(Device);
    m_PressedImage = new CTexture(Device);
    m_Pressed = false;
    m_Caption = NULL;
}
```

## 10.5.2 Setting Pressed and Unpressed Images

One of the first things an application should do with class `CXButton` is set its pressed and unpressed images. These are the images used for its background, depending on its pressed status. These can be set using the `SetPressedImage` and `SetUnpressedImage` methods of `CXButton`. Their definitions are given as follows.

```
bool CXButton::SetUnpressedImage(char* File)
{
    D3DXIMAGE_INFO Info;

    if(m_DefaultImage)
        if(SUCCEEDED(D3DXGetImageInfoFromFile(File, &Info))
        {
            SetWidthHeight(Info.Width, Info.Height);
            if(SUCCEEDED(m_DefaultImage->LoadFromFile(File)))
                return true;
        }

    return false;
}

bool CXButton::SetPressedImage(char* File)
{
    D3DXIMAGE_INFO Info;

    if(m_PressedImage)
        if(SUCCEEDED(m_PressedImage->LoadFromFile(File)))
            return true;

    return false;
}
```



### TIP

Please ensure the `m_Pressed` Boolean member of `CXButton` is set appropriately during `OnMouseDown`, `OnMouseUp`, `OnKeyDown`, and `OnKeyUp` events. If you do not, then you will not know which image to draw as the control is painted.

### 10.5.3 Setting the Button Caption

It is probable that an application will want to set the button's caption. The caption refers to the text shown atop the background, such as "OK" or "Cancel." It often describes the purpose of the button or consequences arising from clicking on it. The caption is actually a `CXLabel` control and its text can be set using the `SetCaption` method of `CXButton`. Notice how a label is created based upon a system font retrieved using some Win32 API functions. See the Note below for more information on those.

```
void CXButton::SetCaption(char* Caption)
{
    LOGFONT lf;

    if(m_Caption)
        delete m_Caption;

    SystemParametersInfo(SPI_GETICONTITLELOGFONT, sizeof(lf), &lf, 0);
    m_Caption = new CXLabel(lf, GetDevice());
    m_Caption->SetWidthHeight(0,0);
    m_Caption->SetCaption(Caption);
}
```



#### NOTE

This function retrieves the system font information to pass on to `ID3DXFONT` using `SystemParametersInfo`. Its declaration is below. If you are unsure how to use such functions, please consult the Windows SDK documentation. See the code below for more information.

```
BOOL SystemParametersInfo
(
    UINT uiAction,
    UINT uiParam,
    PVOID pvParam,
    UINT fWinIni
);

typedef struct tagLOGFONT {
```

```

LONG lfHeight;
LONG lfWidth;
LONG lfEscapement;
LONG lfOrientation;
LONG lfWeight;
BYTE lfItalic;
BYTE lfUnderline;
BYTE lfStrikeOut;
BYTE lfCharSet;
BYTE lfOutPrecision;
BYTE lfClipPrecision;
BYTE lfQuality;
BYTE lfPitchAndFamily;
TCHAR lfFaceName[LF_FACESIZE];
} LOGFONT, *PLOGFONT;
    
```

## 10.5.4 Painting

CXButton is no different from any other control in terms of receiving OnRender events whenever it should paint itself. For CXButton individually though, the painting process involves drawing both its appropriate background and caption text if any. The code to achieve this is shown below.

```

bool CXButton::OnRender()
{
    if(m_Pressed)
        SetCanvas(m_PressedImage);
    else
        SetCanvas(m_DefaultImage);

    this->RenderMe();

    if(m_Caption)
    {
        D3DXVECTOR2 Abs;
        Abs.x = 0;
        Abs.y = 0;
        GetAbsolutePosition(&Abs);
    }
}
    
```

```
        m_Caption->SetXYPos((int) (GetWidth()/2), (int)
                           (GetHeight()/2));
        Abs.x += m_Caption->GetXPos();
        Abs.y += m_Caption->GetYPos();

        m_Caption->SetXYPos(Abs.x, Abs.y);
        m_Caption->OnRender(NULL);
    }

    return true;
}
```



---

**NOTE**

CXButton paints CXLable manually. In other words, CXLable was not added as a child control of CXButton. This is because CXLable would have precluded the button from receiving mouse events were the user to click within the button but also with the boundaries of the label.

## 10.5.5 Destructor

CXButton's destructor is simple. It releases all images, deletes the font object, and removes the label class if it's valid. Take a look at its definition below. The next section examines a sample application that continues from the previous chapter and adds both a label and a button.

```
CXButton::~CXButton()
{
    if(m_Caption)
        delete m_Caption;

    DeleteObject(m_Font);

    if(m_DefaultImage)
        delete m_DefaultImage;

    if(m_PressedImage)
        delete m_PressedImage;
}
```



**NOTE**

One thing that was not covered here about buttons was handling which operations should be performed when clicked. Such functionality should be coded during an `OnMouseDown` or `OnMouseUp` event. However, the chances are that you're going to want to use more than one button and that each of them are going to do something different. This can be handled quite simply in a number of different ways. You could code a new button class or add a virtual function to `CXButton` that is invoked in derived classes and overridden to handle a button's action individually.

## 10.6 **CXLabel and CXButton** — **A Sample Application**

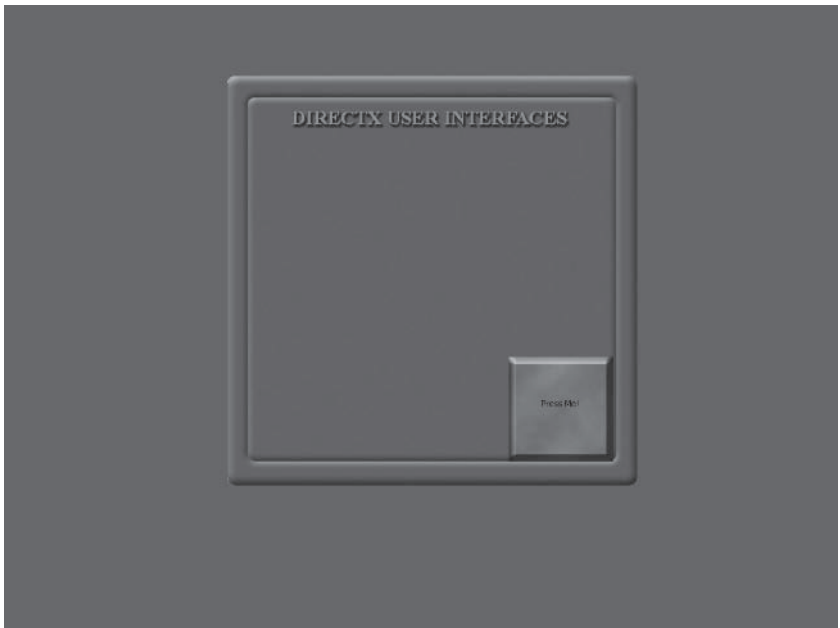


Figure 10.5

This section amends the sample project created in the previous chapter. Until now it used `UI LIB` to present a single window on the screen. Now, this window will contain a label and a button control.

You can follow through and make the changes yourself or load it up from the companion CD. It can be found at Book Code\Part II\Chapter 10. Its code is below. Additions have been highlighted in bold.

```
#include <d3d9.h>
#include <d3dx9.h>
#include <JILIB.H>

LPDIRECT3D9          g_pD3D          = NULL;
LPDIRECT3DDEVICE9    g_pd3dDevice    = NULL;
CXWindow* g_Desktop; //Main desktop window
CXWindow* g_Window;   //A standard window
CXButton* g_Button;   //A button control

//-----
// InitD3D
//-----
HRESULT InitD3D(HWND hWnd)
{
    if(NULL == (g_pD3D = Direct3DCreate9(D3D_SDK_VERSION)))
        return E_FAIL;

    D3DPRESENT_PARAMETERS d3dpp;
    ZeroMemory(&d3dpp, sizeof(d3dpp));
    d3dpp.Windowed = TRUE;
    d3dpp.SwapEffect = D3DSWAPEFFECT_DISCARD;
    d3dpp.BackBufferFormat = D3DFMT_UNKNOWN;

    if(FAILED(g_pD3D->CreateDevice(D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL, hWnd,
                                  D3DCREATE_SOFTWARE_VERTEXPROCESSING,
                                  &d3dpp, &g_pd3dDevice)))
    {
        return E_FAIL;
    }

    g_Desktop = new CXWindow(WINDOW_TYPE_DESKTOP); //Create desktop window
    g_Desktop->SetWidthHeight(640,480); //Make desktop match window size
    g_Desktop->SetXYPos(0,0); //Set desktop to window origin
}
```

```

//Create standard window
g_Window = new CXWindow(WINDOW_TYPE_STANDARD);
g_Window->LoadCanvasFromFile("test.jpg");
g_Window->SetXYPos(0,0);

g_Desktop->AddChildControl(Window); //Set standard window
//as child of desktop

g_Button = new CXButton(g_pd3dDevice);
g_Button->SetXYPos(100,100);
g_Button->SetPressedImage("Pressed.jpg");
g_Button->SetUnpressedImage("Unpressed.jpg");
g_Button->SetCaption("Press Me!");

g_Window->AddChildControl((CXControl*) g_Button);

return S_OK;
}

//-----
// Cleanup
//-----
VOID Cleanup()
{

    delete g_Desktop; //Delete desktop and all children

    if(g_pd3dDevice != NULL)
        g_pd3dDevice->Release();

    if(g_pD3D != NULL)
        g_pD3D->Release();
}

//-----
// Render
//-----
VOID Render()
{
    if(NULL == g_pd3dDevice)

```

```

        return;

    g_pd3dDevice->Clear(0, NULL, D3DCLEAR_TARGET, D3DCOLOR_XRGB(0,0,255),
        1.0f, 0);

    if(SUCCEEDED(g_pd3dDevice->BeginScene()))
    {
        g_Desktop->PostMessage(WM_PAINT,0,0, NULL); //Post WM_PAINT message
        g_pd3dDevice->EndScene();
        g_pd3dDevice->Present(NULL, NULL, NULL, NULL);
    }
}

//-----
// MsgProc
//-----
LRESULT WINAPI MsgProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
    switch(msg)
    {
        case WM_DESTROY:
            Cleanup();
            PostQuitMessage(0);
            return 0;

        case WM_PAINT:
            Render();
            ValidateRect(hWnd, NULL);
            return 0;

        case WM_MOUSEMOVE:
        case WM_LBUTTONUP:
        case WM_LBUTTONDOWN:
        case WM_KEYUP:
        case WM_KEYDOWN:
            //Post other messages
            g_Desktop->PostMessage(msg, wParam, lParam, NULL);
    }
}

```

```

        return 0;

    }

    return DefWindowProc(hWnd, msg, wParam, lParam);
}

//-----
// WinMain – Application starts here
//-----
INT WINAPI WinMain(HINSTANCE hInst, HINSTANCE, LPSTR, INT)
{

    WNDCLASSEX wc = {sizeof(WNDCLASSEX), CS_CLASSDC, MsgProc, 0L, 0L,
                    GetModuleHandle(NULL), NULL, NULL, NULL, NULL,
                    "DirectX User Interfaces", NULL};

    RegisterClassEx(&wc);

    HWND hWnd = CreateWindow("DirectX User Interfaces", "CXWindow",
                            WS_OVERLAPPEDWINDOW, 100, 100, 640, 480,
                            GetDesktopWindow(), NULL, wc.hInstance, NULL);

    if(SUCCEEDED(InitD3D(hWnd)))
    {

        ShowWindow(hWnd, SW_SHOWDEFAULT);
        UpdateWindow(hWnd);

        MSG msg;
        while(GetMessage(&msg, NULL, 0, 0))
        {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
    }

    UnregisterClass("DirectX User Interfaces", wc.hInstance);
    return 0;
}

```

## 10.7 Conclusion

---

This chapter introduced two of the most prevalent controls in an interface, namely labels and buttons. The next chapter continues in a similar vein by investigating the development of two more controls that will form a part of UI LIB. First, however, let's recap what we've learned:

- Labels are rectangular areas inside which text is displayed. These are encapsulated by class `CXLabel`.
- `CXLabel` also encapsulates the `ID3DXFont` interface. This is similar to `ID3DXSprite` and is used to draw text on the screen.
- Buttons are rectangular areas that can have one of two states — pressed or unpressed. Consequently, they have different appearances for each state. Buttons are encapsulated by class `CXButton`.
- Both `CXLabel` and `CXButton` are designed to be child controls of a window.

# Text Boxes and Check Boxes

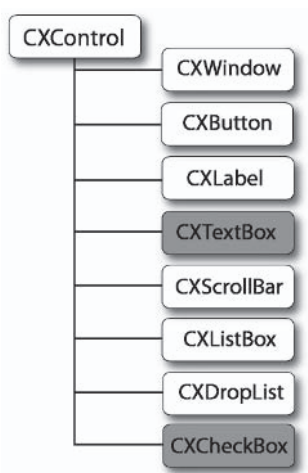


Figure 11.1

UI LIB takes a giant step forward in this chapter as we consider the development of two further controls: text boxes and check boxes. The former control is technically demanding and the latter provides us with some respite. Ultimately, the following topics are discussed in this chapter.

- Text boxes
- Strings and class `std::string`
- The text box caret

- ID3DXLine interface
- Check boxes
- Checked states

## 11.1 Text Boxes and Check Boxes

---

Like their label and button predecessors, text boxes and check boxes are intended to be child controls of a window, windows such as `CXWindow`. Both text boxes and check boxes will be derived from `CXControl`, the former being encapsulated into class `CXTextBox` and the latter into class `CXCheckBox`. The development of these controls is explained throughout this chapter. This process begins in the next section as we consider text boxes.

## 11.2 Text Boxes

---

Text boxes can be a real problem to implement. In fact, they're quite possibly the most technically demanding control of this book. This is primarily because of the nature of text boxes themselves. They store text of variable length, which changes in both content and size as a user types, among other features. The following list contains the key aspects of text boxes and thereby outlines some of the issues to resolve.

- Text boxes are regions inside which text accumulates from user keypresses. Essentially, they are strings that can change in content and length as a user types. The length is increased as a user presses normal character keys and decreased as a user removes characters by deleting them.
- Text boxes use a flashing caret to indicate where the next typed character will be inserted into the string. Often, this will appear at the end. However, users can press the arrow keys or use the mouse to pick new locations. Because it flashes, a caret therefore fluctuates recurrently between two states of visibility —

visible and invisible. Additionally, because a caret iteratively moves through text it effectively has two different positions. These are caret position — the positive integer offset between the beginning and end of the string where the caret resides and where typed text is inserted — and the caret’s absolute position — the actual on-screen position of the caret. This is actually two points representing the top and bottom of the caret line, respectively.

- Text boxes ensure the text’s width and height is maintained. They will not allow the text to extend beyond its geometric boundaries by exceeding its canvas’s width or height by wrapping onto the next line.
- A text box typically only allows its text to be of one size, color, and font.

These points summarize text boxes. However, before implementing `CXTextBox` further, one must learn more about manipulating strings and about drawing lines to represent the caret. These are considered in the following sections as we explore `std::string` and `ID3DXLine`.

## 11.3 Clever Strings — `Std::String`

A text box essentially represents a string of variable length that a user builds through keypresses, by either the insertion or the deletion of characters. Implementing such a string that must grow or shrink dynamically typically involves lots of tiresome memory and string handling code, particularly if you decide to store it by using a standard `char*`. Doing so means you’ll encounter a plethora of different issues, such as how to ensure a string is always of a sufficient length and how to insert characters at appropriate positions without overwriting existing data; and the list just goes on. However, in the previous chapter I promised to demonstrate a superior class for storing and handling strings, a class that performs all such tedium for us. Specifically, I am referring to class

`std::string`; included in `string.h` and bundled with Visual C++. The next few sections examine how to use `std::string` to manipulate strings.

**TIP**

For Borland C++ Builder developers, this class is more or less analogous to type `AnsiString`.

### 11.3.1 Initialization and Assigning



Figure 11.2

Like me, those who first learned C++ probably read or were told that C++ strings should be declared as an array of chars, similar to Figure 11.2. Your code to create and initialize a string would often look something like the following:

```
char* string = new char[STRING_LENGTH];  
ZeroMemory(string, STRING_LENGTH);  
strcpy(string, "hello world\n");
```

There's nothing technically wrong with this code. It is correct. It creates a string of a predetermined size and copies characters into it. However, the string cannot store text above its size limit and wastes trailing space if its text is less than its limit. In effect, it is both inefficient and inconvenient. Alternatively, class `std::string` provides us with a more elegant and versatile solution that holds strings of different lengths and ensures its size adjusts accordingly. The code above could be replaced to use `std::string` as follows. You'll no doubt notice how much better it looks.

```
std::string string = "hello world\n";
```

## 11.3.2 String Lengths

Previously, developers implicitly knew the length of strings because strings were either of a predetermined size or developers used the `strlen` function. Class `std::string` returns the length of any string it holds quite simply using its `length` method.

```
int LengthOfString = string.length();
```

## 11.3.3 Editing and Appending Strings

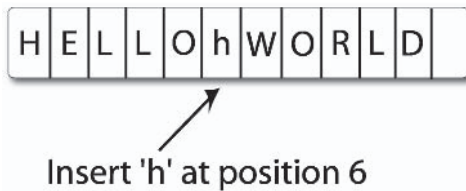


Figure 11.3

Changing a string's contents or appending characters to specific points of a string was often a troublesome issue using `chars`. Not so with `std::string`. Here, developers can completely change a string by simply using the assignment operator or can insert any number of characters at any point of a string by using the `insert` method. This requires a position to insert and a string to place there. The code below demonstrates both methods of string amendment.

```
int STRINGINSERTPOS = 5;

String = "hello world\n";    //String looks like "hello world"
String = "hello";           //String looks like "hello"
String += " world"          //String looks like "hello world"

String.insert(STRINGINSERTPOS, "a new string");
//String looks like "helloa new stringworld"
```

### 11.3.4 Copying Substrings

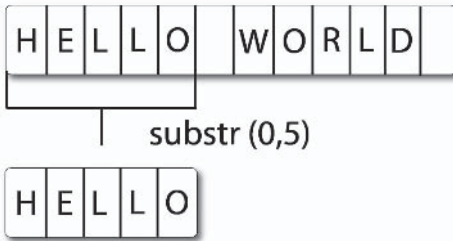


Figure 11.4

One of `std::string`'s other nice features is the ability to copy a specified number of characters from a specific location in a string to a separate `std::string`. This is achieved with the `substr` method. It accepts a starting position where copying begins and the length of characters to copy. These are then returned in a new instance of `std::string` that represents the copied string.

```
std::string Original_string = "hello world\n";  
std::string Copied_String = "";  
  
Copied_String = Original_string.substr(0,5);  
// Copied_String looks like "hello"
```

### 11.3.5 Converting Strings to `char*`

So far, instances of `std::string` cannot be passed as arguments to Windows API functions or others that require arguments of type `char*`. This problem is easily solved by the `cstr` method, which returns a pointer to type `char` representing the beginning of the string. This can be seen below.

```
const char* str = string.cstr();
```

**TIP**

Please note that `cstr` returns type `const char` and therefore cannot be passed to functions that modify the string's contents.

## 11.3.6 Erasing and Emptying

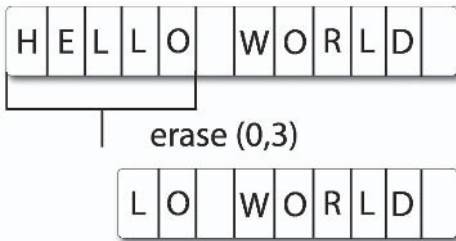


Figure 11.5

Finally, the process of emptying `std::string` is achieved by the `empty` method. This ensures the string is cleaned of all characters, from start to end. The process of erasing is accomplished with the `erase` method. This refers to the removal of a specified quantity of characters, beginning from a defined point inside the string. These functions are demonstrated in the code below.

```
std::string string = "hello world\n";  
string.erase(0,3); //String looks like "lo world\n";  
string.empty();   //String looks like ""
```

## 11.4 Lines — ID3DXLINE

The second issue to consider before finally implementing text boxes is drawing a line to represent the caret. The text box caret represents the flashing line that increments through characters in a text box and indicates where input characters will appear. This line is implemented and drawn by Direct3D's `ID3DXLine` interface, an interface structurally similar to `ID3DXFont` and `ID3DXSprite`. Its

purpose is to draw lines between a series of specified points. This interface is created by the `D3DXCreateLine` function, whose declaration and parameters follow.

```
HRESULT D3DXCreateLine
(
    LPDIRECT3DDEVICE9 pDevice,
    LPD3DXLINE* ppLine
);
```

---

**LPDIRECT3DDEVICE9 pDevice:** Pointer to the Direct3D device that should draw the line.

---

**LPD3DXLINE\* ppLine:** Address to which an instance of `LPD3DXLINE` should be returned.

---

## 11.4.1 Drawing Lines

`ID3DXLine` makes the process of drawing lines quick and simple. To achieve this, developers must call the `Draw` method. It accepts several different parameters, of which an array of `D3DXVECTOR2` structures specifies the points to connect by lines. For a caret, this will simply be two points, the top and bottom of the line. The `Draw` method's function declaration and parameters follow.

```
HRESULT Draw(
    CONST D3DXVECTOR2* pVertexList,
    DWORD dwVertexListCount,
    D3DCOLOR Color
);
```

---

**CONST D3DXVECTOR2\* pVertexList:** Address to a list of `D3DXVECTOR2` structures defining points to connect.

---

**DWORD dwVertexListCount:** The number of points contained in `pVertexList`.

---

**D3DCOLOR Color:** Color of the line.

## 11.5 CTextBox — The Class Declaration

Having examined strings and lines, we can now move onto examining text boxes. These will be encapsulated into class CTextBox. Its full declaration appears below. Don't be concerned if you do not understand all the code at this stage. Subsequent sections examine its implementation in more detail.

```
class CTextBox : public CControl
{
protected:
    std::string m_Text;           //Actual text of the text box
    LPD3DXFONT m_Font;         //Interface to draw text

    LPD3DXLINE m_CaretLine;    //Caret line interface
    D3DXVECTOR2 m_CaretVector[2]; //Top and bottom points of the line
    bool m_CaretVisible;       //Is line visible?
    long CaretCharPos;         //Index of caret

    float m_TextWidth;

public:
    CTextBox(LOGFONT Font, LPDIRECT3DDEVICE9 Device);
    ~CTextBox();
    bool OnRender();
    void OnMouseDown(int Button, int X, int Y);
    void OnMouseMove(int X, int Y);
    void OnMouseUp(int Button, int X, int Y);
    void OnSetFocus();
    void OnLostFocus();
    void OnKeyDown(WPARAM Key, LPARAM Extended);
    void OnKeyUp(WPARAM Key, LPARAM Extended);
    void OnSysKey(WPARAM Key, LPARAM Extended);
    const char* GetText() const {return m_Text.c_str();}
    void SetText(char* Text);
};
```

```

    long GetCharAtPos(int X, int Y);
    bool CursorIntersectChar(int X, int Y);
    FLOAT GetStringWidth(std::string String);
    FLOAT GetStringHeight(std::string String);
    long GetCaretPos() const {return CaretCharPos;}
    void SetCaretPos(long Pos);
    bool InsertText(char* Text);
    long RemoveText(long Quantity);
}

```

## 11.5.1 Constructor

CXTextBox is instantiated by applications whenever they require a text box. The class constructor ensures all its members are set to initial values. This includes creating the line interface to draw the caret, initializing `std::string` to maintain a text box string, and creating a font interface to draw the text.

```

CXTextBox::CXTextBox(LOGFONT Font, LPDIRECT3DDEVICE9 Device) :
    CXControl(Device)
{
    D3DXCreateLine(Device, &m_CaretLine);
    D3DXCreateFont(Device, Font.lfWidth, Font.lfHeight, Font.lfWeight, 1,
        Font.lfItalic, Font.lfCharSet, Font.lfOutPrecision,
        Font.lfQuality, Font.lfPitchAndFamily, Font.lfFaceName,
        &m_Font);

    ZeroMemory(m_CaretVector, sizeof(D3DXVECTOR2) * 2);

    m_TextWidth = 0.0f;
    m_Text = "";
    m_CaretVisible = false;
    CaretCharPos = 0;
}

```

## 11.5.2 Text Width and Height

The width of text inside a text box, in terms of pixels, should be less than or equal to its canvas's width, but not greater. In other words, the text must not stretch beyond the control's boundaries. Conversely, a caret should be equal to the height of the text and not less than or greater than. Hence, it is important to know the width and height of text inside the text box. To compute these values we use the DrawText method of ID3DXFont. Instead of passing it normal flags to draw text, we can also pass it a DT\_CALCRECT constant to size a RECT structure according to the specified text, font, and size. The following two methods of class CXTexBox, GetStringWidth and GetStringHeight, have been added to calculate and return the relevant size of the specified text.

```

FLOAT CXTexBox::GetStringWidth(std::string String)
{
    RECT String_Info;
    ZeroMemory(&String_Info, sizeof(RECT));

    if(m_Font)
        m_Font->DrawText(NULL, String.c_str(), String.length(),
                        &String_Info, DT_CALCRECT, D3DCOLOR_XRGB(0,0,0));

    return String_Info.right;
}

```

```

FLOAT CXTexBox::GetStringHeight(std::string String)
{
    RECT String_Info;
    ZeroMemory(&String_Info, sizeof(RECT));

    if(m_Font)
        m_Font->DrawText(NULL, String.c_str(), String.length(),
                        &String_Info, DT_CALCRECT, D3DCOLOR_XRGB(0,0,0));

    return String_Info.bottom;
}

```

### 11.5.3 Setting Text

One of the first things an application will want to do with a text box's text is completely reinitialize it to some starting text, quite often just a blank string. Assignment of text is achieved through the `SetText` method. The code for this is shown below. Notice how it rejects strings whose length is greater than its canvas's width in pixels. Upon acceptance it also stores the current text width in member `m_Text`. The pixel width of text will be important later, as we attempt to compute how big a step the caret should increment, etc.

```
void CTextBox::SetText(char* Text)
{
    Width = GetStringWidth(m_Text);

    if(Width <= GetWidth())
    {
        m_Text = Text;
        m_TextWidth = Width;
    }
}
```

### 11.5.4 Text Box Caret



On Right Arrow Key Press (Caret + 1)

Figure 11.6

The text box caret is the flashing line that indicates where new characters are inserted. It has been mentioned that it has two states of visibility — visible and invisible. This is managed by the `m_CaretVisible` member. It also has two positions: an offset into the text box string and a geometric position on-screen. These are

recorded by the `m_CaretPos` and `m_CaretPosition` members. Usually a developer specifies a caret position in terms of its former position, that of an index into the string. For example, you'll want to position the caret at the third character in the text box. The following `SetCaretPos` method of `CXTextBox` sets the caret position to a specified location along its string and updates both positions accordingly.

```
void CXTextBox::SetCaretPos(long Pos)
{
    if((Pos >= 0) && (Pos <= m_Text.length()))
    {
        std::string tmpstr = m_Text.substr(0, Pos);

        //Top of line
        m_CaretVector[0].x = GetStringWidth(tmpstr); //Set X Pos

        //Bottom of line
        m_CaretVector[1].x = m_CaretVector[0].x;

        CaretCharPos = Pos;
    }
}
```



#### **NOTE**

The Y position of the caret was not set here because it remains constant.

## 11.5.5 Inserting Text

Inserting text at the caret position entails a number of distinct stages. This is achieved with the `InsertText` method of `CXTextBox`. Broadly, it inserts text into `std::string` at the caret position and then increments the caret by however many characters were inserted, usually one. Specifically, this method is invoked during the `OnKeyDown` event to insert the pressed character at the caret location. Take a look at `InsertText`'s function definition.

```
bool CXTexBox::InsertText(char* Text)
{
    std::string TmpStr = Text;
    if((m_TextWidth + GetStringWidth(TmpStr)) <= GetWidth())
    {
        m_Text.insert(CaretCharPos, Text);
        m_TextWidth = GetStringWidth(m_Text);
        SetCaretPos(GetCaretPos()+strlen(Text));
        return true;
    }

    return false;
}
```

## 11.5.6 Removing Text

Typically, as a user presses the Backspace key you're going to want to delete the character immediately preceding the caret position. CXTexBox can remove a specified number of characters appearing before the caret using the RemoveText method. This process is more or less the reverse of inserting characters, as the following definition demonstrates.

```
long CXTexBox::RemoveText(long Quantity)
{
    SetCaretPos(GetCaretPos()-Quantity);
    m_Text.erase(CaretCharPos, Quantity);
    m_TextWidth = GetStringWidth(m_Text);
    return m_TextWidth;
}
```

## 11.5.7 Processing Keypresses

Generally, when a user presses a key, characters accumulate inside the text box, but if it is a control key, an operation should be performed instead. For example, if the user presses Delete or Backspace, a character should be removed, or if the user presses

the left and right arrows, then the caret increments and decrements through the text box's string. The following code lists the `OnKeyDown` event of `CXTextBox` and demonstrates how to handle various keypresses.

```
void CXTextBox::OnKeyDown(WPARAM Key, LPARAM Extended)
{
    switch (Key)
    {
        case VK_BACK:
        case VK_DELETE:
            {
                RemoveText(1);
            }
            break;

        case VK_LEFT:
            {
                SetCaretPos(GetCaretPos()-1);
            }
            break;

        case VK_RIGHT:
            {
                SetCaretPos(GetCaretPos()+1);
            }
            break;

        case default:
            {
                InsertText((char*) &Key);
            }
    }
}
```

## 11.5.8 Cursor Positioning

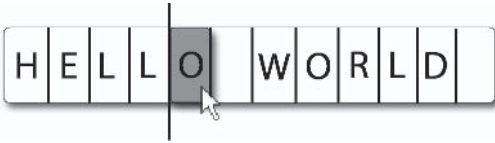


Figure 11.7

Cursor positioning allows a user to adjust the caret position using the mouse as well as the keyboard arrow keys. In other words, the caret can be positioned by the cursor. However, this requires more code than is currently implemented. Typically, you know the caret should be positioned according to the mouse cursor whenever the user clicks the left mouse button, as long as the cursor intersects the text's bounding rectangle. Remember, the text's bounding rectangle can be smaller or equal to the text box's overall area but not greater. Its leftmost extent precedes the first character and its rightmost extent ends at the last character. The following `CursorIntersectChar` method has been added to `CXTextBox` to determine whether the cursor intersects the text box's text and returns `TRUE` or `FALSE` accordingly. The next section demonstrates how to deduce the exact position indicated by the cursor.

```
bool CXTextBox::CursorIntersectChar(int X, int Y)
{
    D3DXVECTOR2 Pos;

    this->GetAbsolutePosition(&Pos);

    if((X >= Pos.x) && (X <= Pos.x + m_TextWidth))
        if((Y >= Pos.y) && (Y <= Pos.y + GetHeight()))
            return true;

    return false;
}
```

## 11.5.9 Caret at Cursor

CXTextBox can now determine whether the cursor intersected the text's bounding rectangle using the `CursorIntersectChar` method above. If this returns `TRUE`, it means the user clicked on the text box's string. However, it still must determine exactly which position within the string was clicked so the caret can be moved there. Effectively, it must translate the cursor (X,Y) coordinate into a horizontal offset in the string. This process is performed by CXTextBox's `GetCharAtPos` method, which returns the indicated string offset. Take a look at its function definition below.

```
long CXTextBox::GetCharAtPos(int X, int Y)
{
    long Pos = 0;
    long Left = 0;
    long Right = 0;

    D3DXVECTOR2 Abs;
    ZeroMemory(&Abs, sizeof(D3DXVECTOR2));
    GetAbsolutePosition(&Abs);

    Left = Abs.x;

    while(Pos < m_Text.length())
    {
        std::string TmpStr = m_Text.substr(Pos, 1);

        Right = GetStringWidth(TmpStr);

        if((X >= Left) && (X <= Left + Right))
            break;

        Pos++;
        Left += Right;
    }

    return Pos;
}
```

## 11.5.10 Handling the Mouse

Having coded methods to process mouse input and translate cursor positions, `CXTextBox`'s `OnMouseDown` event can now be defined as follows to successfully position the caret at the cursor.

```
void CXTextBox::OnMouseDown(int Button, int X, int Y)
{
    if(m_Focus)
        if(CursorIntersectChar(X, Y))
            SetCaretPos(GetCharAtPos(X, Y));
}
```

## 11.5.11 Painting

Finally, the text box can be painted — text, caret, and all. Currently, the text box doesn't have a background loaded onto its canvas, but a `LoadCanvasFromFile` procedure could be added to achieve this, much as demonstrated in Chapter 9. Take a look at the function definition below and notice how the interfaces come into play, specifically those of `ID3DXFont` and `ID3DXSprite`.

```
bool CXTextBox::OnRender()
{
    D3DXVECTOR2 Abs;
    RECT Rct;

    ZeroMemory(&Abs, sizeof(D3DXVECTOR2));

    GetAbsolutePosition(&Abs);

    Rct.left = Abs.x;
    Rct.top = Abs.y;
    Rct.right = Abs.x + GetWidth();
    Rct.bottom = Abs.y + GetHeight();

    m_Font->DrawText(m_Text.c_str(), m_Text.length(), &Rct, DT_LEFT,
        D3DCOLOR_XRGB(255,255,255));
}
```

```

if(m_Focus)
    if(m_CaretVisible)
    {

        D3DXVECTOR2 Absolute[2];
        ZeroMemory(Absolute, sizeof(D3DXVECTOR2) * 2);

        Absolute[0].x = Abs.x + m_CaretVector[0].x;
        Absolute[0].y = Abs.y + m_CaretVector[0].y;
        Absolute[1].x = Abs.x + m_CaretVector[0].x;
        Absolute[1].y = Abs.y + m_CaretVector[0].y + GetHeight();

        m_CaretLine->Draw(Absolute, 2, D3DCOLOR_XRGB(255,255,255));

        m_CaretVisible = false;
    }
    else
        m_CaretVisible = true;

return true;
}

```

## 11.5.12 Cleaning Up

Whew! The text box class was quite demanding, but as its development comes to an end we can code its destructor to clear its members quite simply by using the following code. Thankfully, the remaining sections of this chapter are far simpler as we inspect check boxes.

```

CXTexBox::~CXTexBox()
{
    if(m_CaretLine)
        m_CaretLine->Release();

    if(m_Font)
        m_Font->Release();
}

```

## 11.6 Check Boxes

---

The final control to consider in this chapter is the check box. These are hybrid controls containing a button and a label, and their purpose is to represent one of two states — true or false, checked or unchecked. The following are some of the key points about check boxes.

- Check boxes implicitly use button controls to show a box that can be checked or unchecked.
- Check boxes also use a label control to annotate themselves. This often indicates the field value being set, such as Married, Smoker, Children, etc.
- Check boxes can represent only one of two values and this must correspond to whether it is checked or unchecked. Therefore, this value can either be true or false.

## 11.7 CXCheckBox — The Class Declaration

---

Check box controls are encapsulated in class `CXCheckBox`. It's a simple class whose work is mainly achieved by previously developed controls, namely buttons and labels. Take a look at its class declaration below.

```
class CXCheckBox : public CXControl
{
protected:
    bool m_Checked;
    CXButton* m_TickBox;
    CXLabel* m_Label;
public:
    CXCheckBox(LPDIRECT3DDEVICE9 Device);
    ~CXCheckBox();
    bool GetCheckedState() const {return m_Checked;}
    void SetCheckedState(bool State) {m_Checked = State;}
    bool LoadCheckedImage(char* File);
```

```

bool LoadUncheckedImage(char* File);
void SetCaption(char* Text);
const char* GetCaption() {return m_Label->GetCaption();}
bool OnRender();
void OnMouseDown(int Button, int X, int Y);
void OnMouseMove(int X, int Y);
void OnMouseUp(int Button, int X, int Y);
void OnSetFocus();
void OnLostFocus();
void OnKeyDown(WPARAM Key, LPARAM Extended);
void OnKeyUp(WPARAM Key, LPARAM Extended);
void OnSysKey(WPARAM Key, LPARAM Extended);
};

```

## 11.7.1 Image and Text Loading

Some of the first operations that applications perform with `CXCheckBox` are setting its caption and loading the checked and unchecked images. These may be a check inside a square, a dot inside a circle, or any other representation you choose. The following methods of `CXCheckBox` achieve these operations: `SetCaption`, `LoadCheckedImage`, and `LoadUncheckedImage`. Their definitions follow.

```

bool CXCheckBox::LoadCheckedImage(char* File)
{
    m_TickBox->SetPressedImage(File);
    return true;
}

bool CXCheckBox::LoadUncheckedImage(char* File)
{
    m_TickBox->SetDefaultImage(File);
    return true;
}

void CXCheckBox::SetCaption(char* Text)
{
    m_Label->SetCaption(Text);
}

```

## 11.7.2 Checking and Unchecking

Checking a check box refers to setting its checked status to true and unchecking designates setting it to false. Correspondingly, a check mark or some other icon shall appear or disappear to indicate this. The previous section demonstrated how to load those respective images. The following code shows `CXTextBox`'s `OnMouseDown` event. This determines whether the check box is checked or not.

```
void CXCheckBox::OnMouseDown(int Button, int X, int Y)
{
    if(m_Checked)
    {
        m_TickBox->OnMouseUp(Button, X, Y);
        m_Checked = true;
    }
    else
    {
        m_TickBox->OnMouseDown(Button, X, Y);
        m_Checked = false;
    }
}
```

## 11.7.3 Painting

Painting a check box occurs during the render event, like all other controls. For `CXCheckBox`, the painting process is particularly simple. It merely involves invoking the render events in its button and label members, as follows.

```
bool CXCheckBox::OnRender()
{
    D3DXVECTOR2 Abs;
    ZeroMemory(&Abs, sizeof(D3DXVECTOR2));

    GetAbsolutePosition(&Abs);
```

```

m_TickBox->SetXYPos(Abs.x + 0,Abs.y);
m_TickBox->OnRender();

m_Label->SetXYPos(Abs.x + m_TickBox->GetWidth(), Abs.y);
m_Label->OnRender();

return true;
}

```

## 11.7.4 Cleaning Up

The final part of CXTexTBox is deleting its members in the constructor. This is another simple issue that needs no explanation. Take a look at the following code.

```

CXCheckBox::~CXCheckBox()
{
    if(m_TickBox)
        delete m_TickBox;

    if(m_Label)
        delete m_Label;
}

```

## 11.8 Conclusion

In completing this chapter you have probably reached the peak of a learning curve in terms of interface development. There is still much work to do, however. The next chapter sees the completion of UI LIB as we consider the highly problematic scroll boxes, list boxes, and drop-down lists. Before moving on, however, let's review the key points of this chapter:

- Text boxes are typically rectangular regions that accumulate text as the user types.

- A text box must process ordinary keypresses and control characters to ensure its internal string adds and removes characters appropriately.
- Text boxes maintain a caret to represent an offset into the string and indicate at which position characters are to be inserted and removed.
- A check box is a hybrid control consisting of a button and a label.
- Check boxes represent one of two states — true or false, checked or unchecked.

# Scrolling Lists

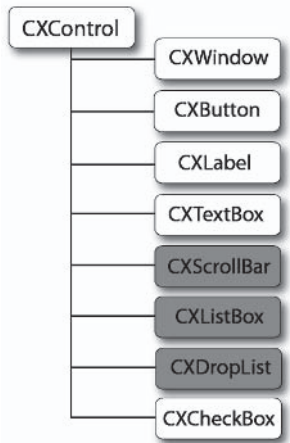


Figure 12.1

This chapter marks the end of Part II, the final stage in developing UI LIB, and introduces three of the more troublesome controls — scroll bars, list boxes, and drop-down lists. This chapter explains how to do the following:

- Create a working scroll bar.
- Scale and tile specific parts of a control.
- Create a working list box.
- Attach a scroll bar for use with other controls.
- Create a working drop-down list.

## 12.1 Scroll Bars, List Boxes, and Drop-Down Lists

---

Scroll bars, list boxes, and drop-down lists are all derived from `CXControl` and are all intended to be child controls of a window. Scroll bars are encapsulated into `CXScrollBar`, list boxes into `CXListBox`, and drop-down lists into — not surprisingly — `CXDropDownList`. Like `CXControl`, however, we will see that scroll bars are not typically instantiated to work on their own, although that is possible. Usually, they are partnered to work as part of a list box, drop-down list, or some other scrollable control. The following sections describe the implementation of each of these controls, starting with `CXScrollBar`.

## 12.2 `CXScrollBar` — Scroll Bars as a Class

---

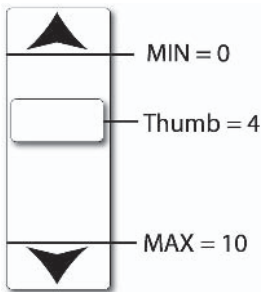


Figure 12.2

A scroll bar is one of those controls that people use every day, whether writing a letter in Microsoft Word or coding the next best-selling product in Microsoft Visual C++. It ultimately is a horizontal or vertical arrangement of two buttons positioned at both ends of the control and which sandwich a movable block, called a thumb. This can slide bidirectionally between each end, and the buttons control its movement toward their respective direction.

The scroll bar's purpose is to provide a convenient solution to display data that cannot all logically fit onto the screen. It does so by simply allowing the user to scroll through the data, moving information into view while other information moves out of view. The following code lists the somewhat intimidating class declaration for `CXScrollBar` and the list that follows discusses the key features of a scroll bar in general. Subsequent subsections provide a detailed examination of this class's implementation.

```
class CXScrollBar : public CXControl
{
private:
protected:
    int m_Min;
    int m_Max;
    int m_ThumbPosition;
    int m_Change;
    FLOAT m_BackgroundHeight;
    bool m_DragMode;
    CXButton* m_Thumb;
    D3DXVECTOR2 m_ThumbPos;
    D3DXVECTOR2 m_ThumbScale;
    FLOAT m_ThumbHeight;
    CXButton* m_RightTopArrow;
    CXButton* m_LeftBottomArrow;
    FLOAT AutoSizeThumb();
    FLOAT AutoSizeBackground();
    void UpdateScaling();
public:
    CXScrollBar(LPDIRECT3DDEVICE9 Device);
    ~CXScrollBar();
    bool OnRender();
    void OnMouseDown(int Button, int X, int Y);
    void OnMouseMove(int X, int Y);
    void OnMouseUp(int Button, int X, int Y);
    void OnSetFocus();
    void OnLostFocus();
    void OnKeyDown(WPARAM Key, LPARAM Extended);
    void OnKeyUp(WPARAM Key, LPARAM Extended);
    void OnSysKey(WPARAM Key, LPARAM Extended);
```

```
void LoadThumbImage(char* File1, char* File2);  
void LoadRightTopArrow(char* File1, char* File2);  
void LoadLeftBottomArrow(char* File1, char* File2);  
void SetValue(int Value);  
int GetValue() {return m_ThumbPosition;}  
int GetPosValue(FLOAT Pos);  
void SetMinMax(int Min, int Max);  
void LoadBackground(char* File);  
void SetChange(int Change) {m_Change = Change;}  
};
```

- Scroll bars can be aligned vertically or horizontally. The example in this book examines the vertical form only, but `CXScrollBar` requires very little adaptation to allow both arrangements.
- Scroll bars have two arrow buttons, one at each end (`m_RightTopArrow` and `m_LeftBottomArrow`). These define the furthest extents of the control.
- Between the buttons is the scroll bar's thumb — the lifeblood of the control. This is a movable block that can exist anywhere between the two buttons and can slide either way according to which button is pressed. The distance between the two buttons is known as the scroll range. For example, in a vertical scroll bar, clicking the up arrow will move the thumb nearer to the top along the scroll range or track.
- Because one scroll button is on one side of the scroll range and another is on the other side, and because the thumb can be anywhere in between, scroll bars conceptually define a single position among a specific range of values, specifically the position of the thumb between either of the two buttons. Thus, one arrow defines a scroll bar's min position (`m_Min`), the other defines its max position (`m_Max`), and the thumb position is an integer between these (`m_ThumbPosition`).
- As one of the scroll buttons is clicked, the thumb moves along the scroll range toward the button's direction. The quantity it steps is known as the change value (`m_Change`). This usually varies proportionally to the min and max values. For scroll bars with a min of 0 and a max of 10, the change will often be 1.

## 12.2.1 The Class Constructor

Applications and controls that create instances of `CXScrollBar` invoke its constructor. This sets all of its members to starting values, as listed below.

```
CXScrollBar::CXScrollBar(LPDIRECT3DDEVICE9 Device) : CXControl(Device)
{
    m_Thumb = new CXButton(Device);
    m_RightTopArrow = new CXButton(Device);
    m_LeftBottomArrow = new CXButton(Device);

    m_ThumbPosition = 0;
    m_ThumbHeight = 0;
    m_Change = 1;
    SetMinMax(0,10);
    SetValue(0);
    m_DragMode = false;

    ZeroMemory(&m_ThumbPos, sizeof(D3DXVECTOR2));
}
```

## 12.2.2 Arrows, a Thumb, and a Background

One of the first things an application should do with a newly created scroll bar is load images for its arrows, its thumb, and its background. It does this using `LoadThumbImage`, `LoadRightTopArrow`, `LoadLeftBottomArrow`, and `LoadBackground`. In this context, a background refers to the image displayed behind the thumb and spans the area of the scroll bar's track, between the buttons. The definitions for these simple methods can be viewed below.

```
void CXScrollBar::LoadThumbImage(char* File1, char* File2)
{
    m_Thumb->SetDefaultImage(File1);
    m_Thumb->SetPressedImage(File2);
    m_ThumbHeight = m_Thumb->GetHeight();
}

//-----

void CXScrollBar::LoadRightTopArrow(char* File1, char* File2)
{
    m_RightTopArrow->SetDefaultImage(File1);
    m_RightTopArrow->SetPressedImage(File2);
}

//-----

void CXScrollBar::LoadLeftBottomArrow(char* File1, char* File2)
{
    m_LeftBottomArrow->SetDefaultImage(File1);
    m_LeftBottomArrow->SetPressedImage(File2);
}

//-----

void CXScrollBar::LoadBackground(char* File)
{
    GetCanvas()->LoadFromFile(File);
    D3DXIMAGE_INFO Info;
    ZeroMemory(&Info, sizeof(D3DXIMAGE_INFO));
    D3DXGetImageInfoFromFile(File, &Info);
    m_BackgroundHeight = Info.Height;
}
```

**NOTE**

One question that might pop into your head at this point is: What happens if the image I want to use for the background is a different size than the scroll bar? This is quite likely considering how narrow scroll bars are. However, this is not a problem. Later sections demonstrate how such images can be programmatically resized and tiled to fit the scroll bar's range track.

## 12.2.3 Width and Height, Min and Max

Another issue that takes priority when initializing the scroll bar is setting its width, height, min, and max values. The min and max represent the greatest extents to which the thumb position can equal. Often, they will be 0 to 10 or 0 to 100. Therefore, the thumb can be anything in between, including those values, but not greater than max or less than min.

The width and height can be set simply by using `CXControl`'s standard `SetWidthHeight` method, or the values can be set individually using `SetWidth` or `SetHeight`. The min and max values can be set with an equally simple method, `SetMinMax`, which is part of `CXScrollBar` and whose definition is below.

```
void CXScrollBar::SetMinMax(int Min, int Max)
{
    m_Min = Min;
    m_Max = Max;

    if(m_ThumbPosition > m_Max)
        m_ThumbPosition = m_Max;

    if(m_ThumbPosition < Min)
        m_ThumbPosition = Min;
}
```

## 12.2.4 Screen Positions to Scroll Values

Sooner or later the user is going to be allowed to set the thumb's position according to a location he clicks on within the scroll bar's range track. In other words, rather than clicking the up and down buttons, the user can click in the scroll range and the thumb should jump to that spot. Consequently, when the user clicks the mouse, you will need to translate the mouse's (X,Y) screen coordinate into an integer that corresponds to the value of the position clicked, somewhere between min and max. Effectively, this process involves dividing the pixel length of the scroll region by `CXScrollBar`'s max value in order to establish the length of each division.

You then must categorize the mouse click by testing which division intersects the cursor. The following `GetPosValue` method of `CXScrollBar` performs this process.

```
int CXScrollBar::GetPosValue(FLOAT Pos)
{
    if((Pos >= m_RightTopArrow->GetHeight())
    if(Pos < (GetHeight() - m_LeftBottomArrow->GetHeight()))
        {
            FLOAT Range = (GetHeight() - m_LeftBottomArrow->GetHeight()) -
                m_RightTopArrow->GetHeight();

            FLOAT Increment = Range / m_Max;

            for(int counter = m_Min; counter < m_Max; counter++)
            {
                FLOAT Item = m_RightTopArrow->GetHeight() + (Increment *
                    counter);

                if(((Pos >= Item) && (Pos <= (Item + Increment))))
                    return counter;
            }
        }

    return 0;
}
```



---

**NOTE**

You will notice that this function does not accept X and Y, just Y. This is because we are dealing with vertical scroll bars only. For horizontal ones, this value will be X instead.

## 12.2.5 Scaling the Thumb

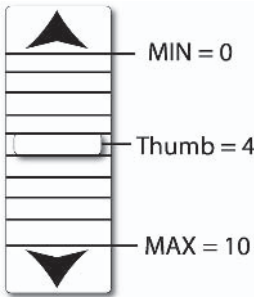


Figure 12.3

Just as there are scaling issues related to the background, there are also scaling issues related to the thumb. As mentioned, the thumb refers to the movable block that slides along the track, between either button, and represents a value between min and max. Thus, the thumb cannot be larger than the height and width of the scroll bar. It must also be scaled dynamically to equal the size of one division along the scroll track — in other words, equal to the pixel area that one unit, out of a total of max, occupies. Making the thumb proportional to the total in this manner will ensure that it scrolls correctly, division by division. The exact process of scaling is handled later in the `OnRender` event by the `ID3DXSprite` interface, which is wrapped in `CXTexture`. This can take care of the intricacies of scaling for us. However, for it to do so, we must compute a scaling factor. This is a floating-point value indicating the factor by which the thumb's true size must increase or decrease. For example, a value of 2 doubles its size, whereas 0.5 halves its size. To compute a scaling factor to size the thumb to one unit along the scroll region, the following method has been added.

```
FLOAT CxScrollBar::AutoSizeThumb()
{
    FLOAT Range = (GetHeight() - m_LeftBottomArrow->GetHeight()) -
                  m_RightTopArrow->GetHeight();

    FLOAT Increment = Range / m_Max;

    return (Increment / m_Thumb->GetHeight()); //The scaling factor
}
```

 **TIP**

Always scaling the thumb to equal the size of one division may not be the best solution since its size is never capped. Thus, it could become too small to grab when its max value is particularly high.

## 12.2.6 Setting the Thumb Position

Once the thumb has been scaled to one division along the scroll region, its position can be set quite simply as an offset from the min value. So, you could specify the value of 5 and the thumb would jump to the fifth unit along the scroll region. Setting the thumb position is something that is going to occur as the user clicks either arrow button, drags the thumb along the scroll region, or clicks a point on the scroll region. Handling such input is examined in the next section. Below, the definition for `SetValue` can be seen; it adjusts the thumb position according to the specified location.

```
void CxScrollBar::SetValue(int Value)
{
    int NewValue = Value;

    if(Value < m_Min)
        NewValue = m_Min;
    else if (Value >= m_Max)
        NewValue = m_Max - 1;

    FLOAT Range = (GetHeight() - m_LeftBottomArrow->GetHeight()) -
                  m_RightTopArrow->GetHeight();
```

```
FLOAT Increment = Range / m_Max;

m_ThumbPos.y = m_RightTopArrow->GetHeight() + (Increment * NewValue);
m_ThumbPosition = NewValue;
}
```

## 12.2.7 Handling Input

The user can position the thumb by clicking either arrow, dragging the thumb, clicking at a specific point in the scroll region, or pressing the keyboard direction arrows. Each of these occurrences is handled in their respective events, and sooner or later end up calling `SetValue` to change the thumb position. The following code lists the `OnMouseDown`, `OnMouseUp`, and `OnMouseMove` events. These demonstrate how to handle scroll bar input.

```
void CXScrollBar::OnMouseDown(int Button, int X, int Y)
{
    D3DXVECTOR2 Abs;
    ZeroMemory(&Abs, sizeof(D3DXVECTOR2));
    GetAbsolutePosition(&Abs);

    if ((Y >= Abs.y) && (Y <= Abs.y + m_RightTopArrow->GetHeight()))
    {
        SetValue(m_ThumbPosition - m_Change);
        return;
    }

    if ((Y >= (Abs.y + GetHeight()) - m_LeftBottomArrow->GetHeight())
        && (Y <= Abs.y + GetHeight()))
    {
        SetValue(m_ThumbPosition + m_Change);
        return;
    }

    FLOAT Thumb = Abs.y + m_ThumbPos.y;

    if((Y >= Thumb) && (Y <= Thumb + m_ThumbHeight))
    {
        m_DragMode = true;
    }
}
```

```

        return;
    }
    FLOAT Va1 = m_RightTopArrow->GetHeight() + (Y -(Abs.y +
        m_RightTopArrow->GetHeight()));

    SetValue(GetPosValue(Va1));
}

//-----
void CXScrollBar::OnMouseMove(int X, int Y)
{
    D3DXVECTOR2 Abs;
    ZeroMemory(&Abs, sizeof(D3DXVECTOR2));
    GetAbsolutePosition(&Abs);

    FLOAT Thumb = (Abs.y + m_RightTopArrow->GetHeight()) + m_ThumbPos.y;

    if(m_DragMode)
    {
        FLOAT Va1 = m_RightTopArrow->GetHeight() + (Y -(Abs.y +
            m_RightTopArrow->GetHeight()));

        SetValue(GetPosValue(Va1));
        return;
    }
}

//-----

void CXScrollBar::OnMouseUp(int Button, int X, int Y)
{
    m_DragMode = false;
}

```

## 12.2.8 Tiling the Background

Section 12.2.5 explained how the thumb can be scaled to fit a specific size and region using a scaling factor; now a similar process must be applied to the background. The background refers to the image shown behind the thumb and spans the entire area of the

scroll region between the two scroll buttons. Typically images will not be of a size that fits this area, particularly since the scroll bar is so narrow. Therefore, to ensure that images of any size fit the entire area, some image manipulation should occur. This process involves two stages: First, scale the background using the same technique to scale the thumb, ensuring it matches the size of a single division along the range track. The image should then be tiled from top to bottom — in other words, drawn recurrently in a tile formation. The following function computes the scaling factor for the background. The second step, tiling, occurs in the next section as we examine the `OnRender` event.

```

FLOAT CXScrollBar::AutoSizeBackground()
{
    FLOAT Range = (GetHeight() - m_LeftBottomArrow->GetHeight()) -
                  m_RightTopArrow->GetHeight();

    FLOAT Increment = Range / m_Max;

    return (Increment / m_BackgroundHeight);
}
    
```

## 12.2.9 Painting

`OnRender` notifies the scroll bar that it should paint itself. Such a process involves drawing the background, the thumb, and the two arrows according to their positions, alignments, and scales. Take a look at the definition for `OnRender` below, and notice how the background is painted in a loop to achieve tiling.

```

bool CXScrollBar::OnRender()
{
    D3DXVECTOR2 Abs;
    ZeroMemory(&Abs, sizeof(D3DXVECTOR2));
    GetAbsolutePosition(&Abs);

    D3DXVECTOR2 Scaling;
    D3DXVECTOR2 Trans;
    Scaling.x = 1;
    
```

```

Scaling.y = AutoSizeBackground();
Trans.x = Abs.x;
Trans.y = Abs.y + m_RightTopArrow->GetHeight();
GetCanvas()->SetScaling(&Scaling);

    for(int counter = m_Min; counter < m_Max; counter++)
    {
        GetCanvas()->SetTranslation(&Trans);

        GetPen()->DrawTexture(GetCanvas());
        Trans.y += m_BackgroundHeight * Scaling.y;
    }

    m_RightTopArrow->SetXYPos(Abs.x, Abs.y);
    m_RightTopArrow->OnRender();
    m_LeftBottomArrow->SetXYPos(Abs.x, Abs.y + (GetHeight() -
                                m_LeftBottomArrow->GetHeight()));
    m_LeftBottomArrow->OnRender();

    UpdateScaling();
    m_Thumb->SetXYPos(Abs.x, Abs.y + m_ThumbPos.y);
    m_Thumb->OnRender();

    return true;
}

```

## 12.2.10 CXScrollBar — Cleaning Up

Finally, the destructor is called as the class is deleted. As usual, it frees whichever objects it allocated memory for. Its definition can be seen below. Following sections examine how the scroll bar class is attached to other controls — namely CXListBox and CXDropDownList — to achieve scrolling.

```

CXScrollBar::~CXScrollBar()
{
    if(m_Thumb)
        delete m_Thumb;

    if(m_RightTopArrow)

```

```
delete m_RightTopArrow;  
  
if(m_LeftBottomArrow)  
    delete m_LeftBottomArrow;  
}
```

## 12.3 List Boxes and List Items

---

Time to develop the next control, list boxes. Now, if you're not very careful, list boxes, and the rest of the list controls for that matter, can become what's technically termed "a real pain in the rear end." So you've got to keep your wits about you. Essentially, list boxes are containers that hold a collection of items, called list items. Typically, the user can use a scroll bar to browse the items and can select one or many. Often, list boxes are employed to show a list of selectable items to the user, such as different models of cars or different program options, etc. Thus, two classes are effectively being developed here — `CXListBox` to hold the items and `CXListItem` to represent a single item. The following sections investigate the development of a list item class and a list box class.

## 12.4 CXListItem — List Items as a Class

---

List boxes may contain no, one, or many list items. They have two backgrounds, one for their selected appearance and one for non-selected, and they also have a caption and can contain an icon. The class declaration for `CXListItem` appears below. The list that follows points out the key features of a list item.

```
class CXListItem : public CXControl  
{  
private:  
protected:  
    FLOAT m_StretchWidthFactor;  
    FLOAT m_StretchHeightFactor;
```

```
    bool IsSelected;
    std::string m_Caption;
public:
    CXLListItem(LPDIRECT3DDEVICE9 Device);
    ~CXLListItem();
    bool OnRender();
    void OnMouseDown(int Button, int X, int Y);
    void OnMouseMove(int X, int Y);
    void OnMouseUp(int Button, int X, int Y);
    void OnSetFocus();
    void OnLostFocus();
    void OnKeyDown(WPARAM Key, LPARAM Extended);
    void OnKeyUp(WPARAM Key, LPARAM Extended);
    void OnSysKey(WPARAM Key, LPARAM Extended);
    void SetSelect(bool State) {IsSelected = State;}
    void SetItemSize(FLOAT Width, FLOAT Height);
    void SetCaption(char* Caption) {m_Caption = Caption;}
    const char* GetCaption() {return m_Caption.c_str();}
};
```

- List items are rectangular areas that have position, width, and height inside a list box. For the purposes of this book, all list items in a single list box have the same width and height.
- List items can be selected or non-selected (`IsSelected`) and have a background to represent either state. For the list items in this book, the same backgrounds will actually be applied to all list items in a single list box, ensuring they maintain a consistent appearance throughout. Thus, the background images are actually stored and managed by the list box, not the items, and are shared between all of the list items. This also helps to ensure memory is conserved. The only properties the items maintain separately, regarding their backgrounds, are the factors by which it stretches them to fit its canvas (`m_StretchWidthFactor` and `m_StretchHeightFactor`).
- List items *can* have an icon, a small image that describes the item or distinguishes it from others. This book deals only with a list item's text, but it would be a trivial matter to include a small texture member that is rendered when the item is painted.

- List items can also have a caption (`m_Caption`). Like backgrounds, since all items have a caption, the list box, not the item, is the owner of a shared `ID3DXFont` interface that each item uses to draw its text.
- The parent control of a list item is the containing list box.

## 12.4.1 The Class Constructor

As list boxes create list items, `CXListItem`'s class constructor is called. This is a simple method that initializes class members to starting values. The definition for this function can be coded as follows.

```
CXListItem::CXListItem(LPDIRECT3DDEVICE9 Device) : CXControl(Device)
{
    m_StretchWidthFactor = 1.0f;
    m_StretchHeightFactor = 1.0f;
    IsSelected = false;
    m_Caption = "";
}
```

## 12.4.2 Setting Item Size

One of the first things a list box should do to a newly created item is initialize it to a starting size. This will allow `CXListItem` to compute scaling factors to size the backgrounds correctly when the item paints itself. For this book, all items should be the same width and height, and all items should share their background images from the list box. The following function, `SetItemSize`, has been added to class `CXListItem` to perform item initialization.

```
void CXListItem::SetItemSize(FLOAT Width, FLOAT Height)
{
    if(GetParentControl())
    {
        D3DXIMAGE_INFO Info;
        CXListBox* List = NULL;
```

```

ZeroMemory(&Info, sizeof(D3DXIMAGE_INFO));
List = (CListBox*) GetParentControl();

D3DXGetImageInfoFromFile(List->GetBackgroundImage()->GetPath(),
                          &Info);

m_StretchWidthFactor = Width / Info.Width;
m_StretchHeightFactor = Height / Info.Height;

SetWidthHeight(Width, Height);
}
}

```

### 12.4.3 Painting

Setting the selected state of a list item and setting the caption have already been defined as inline functions back in the class declaration. Therefore, it is time to move on to how list items paint themselves. Like all other controls, list items paint themselves during an `OnRender` event. However, for list items, this event is typically called at the discretion of the owning list box whenever it requires items to display themselves. The `OnRender` event can be defined as below. Notice how it calls upon the owning list box to provide `CXTexture` pointers to the background images that should be used for either their selected or non-selected backgrounds. Remember, this is because backgrounds are shared between all items. Notice also how an item uses the list box to provide a shared `ID3DXFont` interface to draw text. The details of `CListBox` are examined throughout subsequent sections of this chapter.

```

bool CListItem::OnRender()
{
    D3DXVECTOR2 Abs;
    ZeroMemory(&Abs, sizeof(D3DXVECTOR2));
    GetAbsolutePosition(&Abs);

    D3DXVECTOR2 Scaling;

```

```

ZeroMemory(&Scaling, sizeof(D3DXVECTOR2));

Scaling.x = m_StretchWidthFactor;
Scaling.y = m_StretchHeightFactor;

CListBox* List = (CListBox*) GetParentControl();

if(IsSelected)
{
    if(List->GetSelectedImage())
    {
        List->GetSelectedImage()->SetTranslation(&Abs);
        List->GetSelectedImage()->SetScaling(&Scaling);
        GetPen()->DrawTexture(List->GetSelectedImage());
    }
}
else
{
    if(List->GetBackgroundImage())
    {
        List->GetBackgroundImage()->SetTranslation(&Abs);
        List->GetBackgroundImage()->SetScaling(&Scaling);
        GetPen()->DrawTexture(List->GetBackgroundImage());
    }
}

if(List->GetFont())
{
    RECT Rct;
    ZeroMemory(&Rct, sizeof(RECT));
    Rct.left = Abs.x;
    Rct.top = Abs.y;
    Rct.right = Abs.x + GetWidth();
    Rct.bottom = Abs.y + GetHeight();
    List->GetFont()->DrawText(m_Caption.c_str(), m_Caption.length(),
        &Rct, DT_LEFT, D3DCOLOR_XRGB(255,255,255));
}

return true;
}
    
```

## 12.5 CListBox — List Boxes as a Class

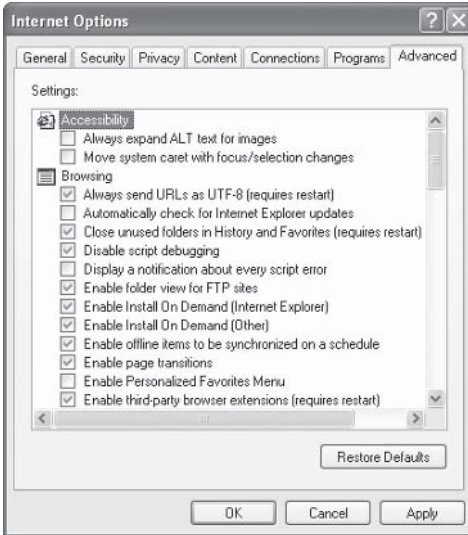


Figure 12.4

A list box is a list of items. Visually, it is a frame or window containing a subset of its items because it may be too small to show them all. It can have no, one, or many list items, and the subset it shows can be controlled by its scroll bar. The scroll bar allows a user to scroll through the list, browsing a specific number of items each time. Take a look at the class declaration for `CListBox` below and then see the list that follows to examine the key points of a list box. These are implemented in following sections.

```
class CListBox : public CXControl
{
protected:
    CListItem* m_ListItems;
    CListItem* m_SelectedItem;
    CTexture* m_Selected;
    CTexture* m_Background;
    LPD3DXFONT m_Font;
```

```

CScrollbar* m_Bar;
bool m_AllowMultiSelect;
long m_ListFrame;
long m_ItemCount;
long ComputeListFrame();
public:
    CListBox(LPDIRECT3DDEVICE9 Device);
    ~CListBox();
    bool OnRender();
    void OnMouseDown(int Button, int X, int Y);
    void OnMouseMove(int X, int Y);
    void OnMouseUp(int Button, int X, int Y);
    void OnSetFocus();
    void OnLostFocus();
    void OnKeyDown(WPARAM Key, LPARAM Extended);
    void OnKeyUp(WPARAM Key, LPARAM Extended);
    void OnSysKey(WPARAM Key, LPARAM Extended);
    void OnItemSelect(CXListItem* Item, int Button, int X, int Y);
    void LoadItemBackgrounds(char* Default, char* Selected);
    CXListItem* AddNewItem(char* Caption);
    void RemoveListItem(CXListItem* Item);
    CTexture* GetBackgroundImage() {return m_Background;}
    CTexture* GetSelectedImage() {return m_Selected;}
    LPD3DXFONT GetFont() {return m_Font;}
    void LoadScrollbar(char* Background, char* Thumb, char* TopArrow,
        char* BottomArrow);
    CXListItem* GetItemAtPos(int X, int Y);
    CXListItem* GetItemByIndex(long Index);
    void ScrollTo(long Value);
    void ClearItems();
    bool ScrollBarIntersect(int X, int Y);
    CXListItem* GetSelected() {return m_SelectedItem;}
    FLOAT GetListWidth() {return GetWidth() - m_Bar->GetWidth();}
};

```

- List boxes can have no, one, or many list items. These are implemented as a linked list and can be added, removed, and edited dynamically. `CXListBox` maintains a pointer to the first list item through member `m_ListItems`.

- Each list item has an index in the list box. This is an integer offset from the beginning of the list, where 0 refers to the first list item.
- Items have the same width and height, share the same two background images for selected and non-selected states, and share the same ID3DXFont interface to render their text. These are created and maintained by CXListBox in members `m_Selected`, `m_Background`, and `m_Font`, respectively.
- Often, list boxes will have more items than they can display. Thus, list boxes must act like a frame, containing only a subset of their items. This frame can be moved by a scroll bar, which can scroll it up and down to view other items. The scroll bar is maintained by `m_Bar`, and CXListBox represents the frame simply with an integer that specifies how many items can be displayed in the list box. This frame is computed by dividing the list box's total height in pixels by the height of one item, remembering that all list items have the same height. The result is stored in member `m_ListFrame`.
- List boxes often implement smooth scrolling such that items at the top and bottom of the frame can be half visible as they slide from view. However, for this book, items scroll on an item-by-item basis; therefore, items can either be visible or not. As each moves out of view, another moves into view.
- List boxes cannot be scrolled in any direction if the item count (`m_ItemCount`) is less than `m_ListFrame`; in other words, if all items can fit within the list box frame. Likewise, list boxes cannot be scrolled upward or downward beyond their top and bottom items, respectively.
- List items can be selected. If the list box allows multi-selection (`m-AllowMultiSelect`), the user may select one or more items together, but if multi-selection is disabled, only one item at a time can be selected. Thus, as one item is selected, another becomes deselected.

## 12.5.1 The Class Constructor

Applications declare instances of `CXListBox` to show a list of items to a user. As the class is created, its constructor is called to initialize its members. The code below shows the definition for this function. Notice how the scroll bar is created and how a font object is assigned. Other members are typically set to `NULL`.

```
CXListBox::CXListBox(LPDIRECT3DDEVICE9 Device) : CXControl(Device)
{
    m_AllowMultiSelect = false;
    m_Selected = NULL;
    m_Background = NULL;
    m_Font = NULL;
    m_Bar = new CXScrollBar(GetDevice());
    m_ListFrame = 0;
    m_ItemCount = 0;
    m_ListItems = NULL;
    m_SelectedItem = NULL;

    LOGFONT Font;
    SystemParametersInfo(SPI_GETICONTITLELOGFONT, sizeof(LOGFONT),
                        & Font, 0);

    D3DXCreateFont(GetDevice(), Font.lfWidth, Font.lfHeight,
                  Font.lfWeight, 1, Font.lfItalic, Font.lfCharSet,
                  Font.lfOutPrecision, Font.lfQuality,
                  Font.lfPitchAndFamily, Font.lfFaceName, &m_Font);
}
```

## 12.5.2 Loading Item Backgrounds

Section 12.4 mentioned how list items have two backgrounds, one to appear if they are selected and one to appear in a default or unselected state. Rather than store these images separately in each item and accumulating goodness knows how much memory, it makes sense to manage these images in the list box and share them among the items. The following function, `LoadItemBackgrounds`, shows how these images can be loaded.

```
void CXListBox::LoadItemBackgrounds(char* Default, char* Selected)
{
    if(!m_Background)
    {
        m_Background = new CXTexture(GetDevice());
        m_Background->LoadFromFile(Default);
    }

    if(!m_Selected)
    {
        m_Selected = new CXTexture(GetDevice());
        m_Selected->LoadFromFile(Selected);
    }
}
```

## 12.5.3 Loading the Scroll Bar

Once item backgrounds have been loaded from image files, the scroll bar should follow. This involves specifying which images should be used for the scroll bar background, the scroll buttons, and the thumb. It also involves setting the (X,Y) position at which the scroll bar should appear within the list box, not to mention setting its width, height, and starting thumb position. The `LoadScrollBar` method has been added to `CXListBox` to achieve this process. It simply requires valid paths to image files.

```
void CXListBox::LoadScrollBar(char* Background, char* Thumb, char* TopArrow,
                              char* BottomArrow)
{
    if(m_Bar)
    {
        m_Bar->LoadThumbImage(Thumb);
        m_Bar->LoadBackground(Background);
        m_Bar->LoadLeftBottomArrow(BottomArrow);
        m_Bar->LoadRightTopArrow(TopArrow);
        m_Bar->SetWidthHeight(32, GetHeight());
    }
}
```

```

        m_Bar->SetXYPos(GetWidth() - m_Bar->GetWidth(), 0);
        m_Bar->SetValue(0);
    }
}
    
```

## 12.5.4 Computing a List Frame

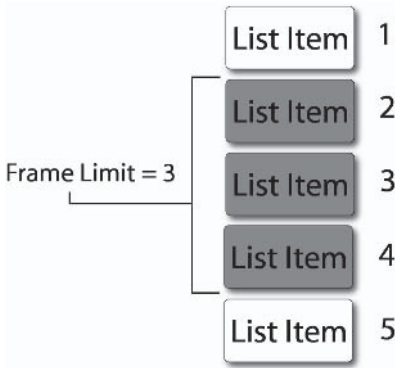


Figure 12.5

A list box can only show a certain number of items based upon its current height and width. If there are more items than can fit, you could either extend the list box’s height and width or show only a subset of items that can be navigated using a scroll bar. We use the second solution here. Thus, computing a list frame refers to the process of determining the maximum number of list items that can be shown inside the list box at once. This is a simple division of the total list box height in pixels by a single item’s height. Later, we’ll be able to determine whether the current number of items can fit in the list box or not, after comparing the item count to the frame maximum. To calculate this maximum, the following `ComputeListFrame` method of `CXListBox` should be used. It will be invoked by other methods later, such as `AddNewItem`, to ensure an updated figure will always be maintained. Notice how it also sets the scroll bar’s min and max values to ensure that the remainder will be scrollable later on.

```
long CListBox::ComputeListFrame()
{
    if(m_ListItems)
    {
        long Frame = (long) GetHeight() / m_ListItems->GetHeight();
        m_Bar->SetMinMax(0, (m_ItemCount+1) - Frame);
        return Frame;
    }
    else
    {
        m_Bar->SetMinMax(0, 0);
        return 0;
    }
}
```

## 12.5.5 Adding List Items

The process of adding a list item is a fairly lengthy one considering the number of different dependencies that must change or in some way be recalculated, such as the list frame, the item count, the item caption, and the item size. Specifically, however, this is really a combination of different processes that have already been demonstrated in previous sections. List items are stored in a linked list and new items are appended to the end. To ensure the item is geometrically positioned in the list correctly, in relation to other items, items are effectively arranged in a tile formation, one above the other. Therefore, a new item's Y position is the cumulative height of all items above it in the list. Take a look at the following `AddNewItem` method of `CListBox` to examine how this process is achieved. Notice that a method called `ScrollTo` is invoked toward the end. This function navigates the list frame to a specific item index, such that the specified item becomes the top of the frame. This method is explained in more detail in a later section.

```
CXListItem* CListBox::AddNewItem(char* Caption)
{
    if((m_Background) && (m_Selected))
    {

        D3DXIMAGE_INFO Info;
        CXListItem* Item; //Item to add

        Item = new CXListItem(GetDevice()); //Create new item
        ZeroMemory(&Info, sizeof(D3DXIMAGE_INFO));
        (m_Background->GetPath(), &Info);

        if(Item)
        {
            Item->SetParentControl(this);
            CXPen* Pen = Item->GetPen();

            if(Pen)
                delete Pen;

            Item->SetPen(GetPen()); //Share parent CXPen

            if(!m_ListItems)
            {
                m_ListItems = Item;
                Item->SetXYPos(0,0);
            }
            else
            {
                CXListItem* Temp = m_ListItems;
                FLOAT Height = 0;

                while(Temp->GetNextSibling())
                {
                    Height += Temp->GetHeight();
                    Temp = (CXListItem*) Temp->GetNextSibling();
                }

                Temp->SetNextSibling((CXControl*) Item);
                Item->SetPreviousSibling((CXControl*) Temp);
            }
        }
    }
}
```

```
        Item->SetXYPos(0,Height + Temp->GetHeight());
    }

    Item->SetItemSize(GetWidth() - m_Bar->GetWidth(), Info.Height);
    Item->SetCaption(Caption);
    m_ItemCount++;
    m_ListFrame = ComputeListFrame(); //How many items can be displayed?
    ScrollTo(0);

    return Item;
}
}
return NULL;
}
```

## 12.5.6 Clearing List Items

There will be times when you'll want to delete the list or repopulate the list. To do this, you'll need to clear the list items. This is achieved by calling `CXListBox`'s `ClearItems` method. Its definition can be coded to delete all items as follows.

```
void CXListBox::ClearItems()
{
    CXListItem* Temp = m_ListItems;

    while(Temp)
    {
        CXListItem* Next = (CXListItem*) Temp->GetNextSibling();

        if(Temp)
            delete Temp;

        Temp = Next;
    }
}
```

## 12.5.7 Getting Items by Index

Each item in a list box has an index. This refers to an integer offset from the list's beginning. The first item has an index of 0, the next item has an index of 1, the next 2, and so on. Thus, occasionally it becomes convenient to select items by index. In other words, you'll want to retrieve a `CXListItem*` pointer to a specific item by specifying an index. The following method of `CXListBox` achieves this.

```
CXListItem* CXListBox::GetItemByIndex(long Index)
{
    long Counter = 0;

    CXListItem* CurrentItem = m_ListItems;

    while(CurrentItem)
    {
        if(Counter == Index)
            return CurrentItem;

        Counter++;
        CurrentItem = (CXListItem*) CurrentItem->GetNextSibling();
    }

    return NULL;
}
```

## 12.5.8 Getting Items by (X,Y) Position

It is likely you're going to want to detect which item a user clicked or distinguish which item resides at a specific screen position. Effectively, you'll need to translate screen coordinates into a corresponding list item. Like determining where the thumb should be positioned on a scroll bar, you must categorize the click by intersection testing — testing an (X,Y) location against the position and area of a list item. The following `GetItemAtPos` method returns a list item according to a screen position.

```

CXLListItem* CXLListBox::GetItemAtPos(int X, int Y)
{
    long Counter = 0;
    D3DXVECTOR2 Abs;
    ZeroMemory(&Abs, sizeof(D3DXVECTOR2));
    GetAbsolutePosition(&Abs);

    CXLListItem* CurrentItem = m_ListItems;

    while(CurrentItem)
    {
        if(CurrentItem->GetVisible())
        if((X >= Abs.x) && (X <= Abs.x + (GetWidth() - m_Bar->GetWidth())))
        if((Y >= Abs.y + CurrentItem->GetYPos()) && (Y <= Abs.y +
            CurrentItem->GetYPos() + CurrentItem->GetHeight()))
            return CurrentItem;

        CurrentItem = (CXLListItem*) CurrentItem->GetNextSibling();
    }

    return NULL;
}

```

## 12.5.9 Scrolling the Frame

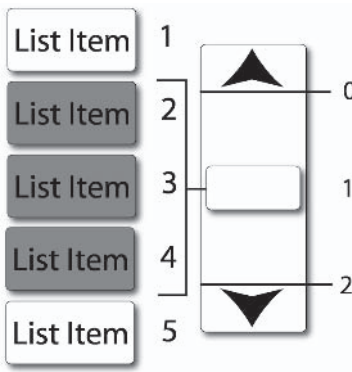


Figure 12.6

If the total number of items in a list box exceeds the maximum frame limit, not all items can be viewed at once. Hence, a scroll bar is required if a user is to navigate all items. Consequently, you'll need a function that connects the scroll bar to the list box, such that certain subsets of list items are visible as long as the scroll bar's thumb is of a certain value. In other words, you want the list to scroll as the scroll bar is moved. One way to achieve this is to perform the following two steps.

1. First, set the scroll bar's min to 0, and set max to equal the number of excess items that cannot fit into the list box's frame; effectively, this value is: `ItemCount` minus `limit`. This allows the user to slide the scroll bar thumb along a distance that can potentially bring every list item into view.
2. Each time the user scrolls the thumb you need to set the list box frame such that a corresponding subset of items comes into view. This is achieved by setting the list item whose index matches the scroll bar's thumb value to be the top item in the list box's frame. In other words, if the thumb equals 5, then the fifth list item should be at the top. All you need to do then is draw every subsequent item until the frame is filled.

To perform this rather complicated scrolling process, the following `ScrollTo` function has been added to `CXListBox`. Take a look at its definition below. It is invoked during `OnMouseUp` events, as we see in the next section, to ensure the list items move according to the scroll bar.

```
void CXListBox::ScrollTo(long Value)
{
    if(m_ItemCount > m_ListFrame) //Can it scroll?
    if((Value + m_ListFrame) <= m_ItemCount)
    {
        //Get first item
        CXListItem* CurrentItem = m_ListItems;

        long Counter = 0;
        float AccumulativeYPos = 0.0f;
```

```
while(CurrentItem)
{

    CurrentItem->SetSelect(false);

    if((Counter >= Value) && (Counter < Value + m_ListFrame))
    //Is it visible?
    {
        CurrentItem->SetVisible(true);
        CurrentItem->SetXYPos(0, AccumulativeYPos);
        AccumulativeYPos+= CurrentItem->GetHeight();

    }
    else
        CurrentItem->SetVisible(false);

    Counter++;

    CurrentItem = (CXListItem*) CurrentItem->GetNextSibling();
}
}
}
```

## 12.5.10 Handling Input

Now that a list box can scroll to a specified index, various input events, such as `OnMouseDown`, `OnMouseUp`, etc., can be coded to process input accordingly. Notice how list items are selected on mouse clicks and how the scroll box moves to particular items as the mouse button is released over the scroll box.

```
void CXListBox::OnMouseDown(int Button, int X, int Y)
{
    CXListItem* Selected = NULL;
    Selected = GetItemAtPos(X, Y);

    if(Selected)
```

```

        OnItemSelect(Selected, Button, X, Y);

        if(ScrollBarIntersect(X, Y)) //True or False; is mouse over scroll bar?
            m_Bar->OnMouseDown(Button, X, Y);
    }

//-----

void CListBox::OnMouseMove(int X, int Y)
{
    m_Bar->OnMouseMove(X, Y);
}

//-----

void CListBox::OnMouseUp(int Button, int X, int Y)
{
    if(ScrollBarIntersect(X, Y))
    {
        m_Bar->OnMouseUp(Button, X, Y);
        ScrollTo(m_Bar->GetValue());
    }
}

```

## 12.5.11 Painting

CListBox must paint itself as it receives OnRender events, like any other control in UI LIB. Here, it must render its scroll bar and list items. The ScrollTo method featured in Section 12.5.9 sets the list box frame and ensures that only visible list items have their visible status set to true. Remember, the visible status was coded into the ancestor control CXControl, not CListBox or CListBoxItem themselves. This makes painting very simple because you simply render only those items that are visible. A mechanism to achieve this process — passing OnRender events to only visible controls — was coded in Chapter 8 and is already part of CXControl. It can be initiated by calling PostToAllReverse and can be used to invoke render events in all visible list items. Take a look at CListBox’s OnRender definition to see how a list box should paint itself. Easy!

```
bool CListBox::OnRender()
{
    D3DXVECTOR2 Abs;
    ZeroMemory(&Abs, sizeof(D3DXVECTOR2));
    GetAbsolutePosition(&Abs);

    if(GetVisible())
    {
        if(m_Bar)
        {
            m_Bar->SetXYPos(Abs.x + (GetWidth() - m_Bar->GetWidth()),
                           Abs.y);
            m_Bar->OnRender();
        }

        if(m_ListItems)
            PostToAllReverse((CXControl*)m_ListItems, WM_RENDER, 0, 0,
                             NULL);
    }
    return true;
}
```

## 12.5.12 Cleaning Up

Finally, after an application has finished using `CListBox`, it should delete the class and thereby call its destructor to free its members. This can be coded as follows.

```
CListBox::~CListBox()
{
    ClearItems();

    if(m_Selected)
        delete m_Selected;

    if(m_Background)
        delete m_Background;
```

```

    if(m_Font)
        m_Font->Release();

    if(m_Bar)
        delete m_Bar;
}
    
```

## 12.6 CXDropDownList — Drop-Down Lists as a Class



Figure 12.7

The final control to examine in this chapter, and in UI LIB, is the drop-down list. A drop-down list is effectively a hybrid control; a form of text box with a list box attached. The user can enter text via normal keypresses or by selecting items from the list. Figure 12.7 demonstrates that a drop-down list is composed of the following three controls: a text box, a list box, and a button that controls the visibility of the list box. Thus, things get simpler because most of the hard work has already been done in previous classes. The following class declaration is given for CXDropDownList. The list that



```

        char* ButtonBackgroundDefault,
        char* ButtonBackgroundSelected,
        char* ItemBackgroundDefault,
        char* ItemBackgroundSelected);
};

```

- A drop-down list is composed of a text box (`m_TextBox`), a list box (`m_ListBox`), and a button (`m_Button`). The button controls whether the list box is visible or not, up or down.
- Accordingly, the drop-down list control will fluctuate between two potential heights, one for when the list is up and one for when it is down (`m_RolledUpHeight`).
- Like text boxes, drop-down lists represent a piece of editable text. This can be built from user keypresses or by selecting items from the list.
- The list box can either be visible or invisible. When visible, the user can select items. Otherwise, it acts like a normal text box (`IsDroppedDown`).

## 12.6.1 The Class Constructor

Applications that instantiate a drop-down list invoke the class constructor. This initializes all its internal members, as with every other class. Its definition is as follows.

```

CXDropDownList::CXDropDownList(LPDIRECT3DDEVICE9 Device) : CXControl(Device)
{
    LOGFONT lf;

    SystemParametersInfo(SPI_GETICONTITLELOGFONT, sizeof(lf), &lf, 0);

    m_ListBox = new CXListBox(Device);
    m_TextBox = new CXTextBox(lf, Device);
    m_Button = new CXButton(Device);

    m_RolledUpHeight = 0.0f;
}

```

## 12.6.2 Initializing the Drop-Down List

Initializing the drop-down list is really a simple process but, because it uses a list box, scroll box, and button, there's a lot of it. You'll need to load images for each of those controls. This could be performed individually, step by step. However, a function called `InitializeDropDownList` has been added to `CXDropDownList`. This will allow applications to perform this rather tedious work in one step. I don't usually recommend coding functions with this many parameters, but here is an example of its desirability. It requires starting height and width values, and paths to all image files that will be used. Take a look at its definition below.

```
bool CXDropDownList::InitializeDropDownList(FLOAT DropDownWidth,
      FLOAT DropDownHeight, FLOAT ListHeight, FLOAT ButtonWidth,
      char* ScrollThumb, char* ScrollTopArrow, char* ScrollBottomArrow,
      char* ScrollBackgroundDefault, char* ButtonBackgroundDefault,
      char* ButtonBackgroundSelected, char* ItemBackgroundDefault,
      char* ItemBackgroundSelected)
{
    D3DXVECTOR2 Abs;
    ZeroMemory(&Abs, sizeof(D3DXVECTOR2));
    GetAbsolutePosition(&Abs);

    SetWidthHeight(DropDownWidth, DropDownHeight);
    m_RolledUpHeight = DropDownHeight;

    m_ListBox->SetXYPos(Abs.x, Abs.y + DropDownHeight);
    m_ListBox->SetWidthHeight(DropDownWidth, ListHeight);

    m_ListBox->LoadScrollBar(ScrollBackgroundDefault,
        ScrollThumb, ScrollTopArrow, ScrollBottomArrow);

    m_ListBox->LoadItemBackgrounds(ItemBackgroundDefault,
        ItemBackgroundSelected);

    m_ListBox->SetVisible(false);
}
```

```
m_Button->SetXYPos(Abs.x + (DropDownWidth-ButtonWidth), Abs.y);
m_Button->SetWidthHeight(ButtonWidth, ListHeight);

m_Button->SetDefaultImage(ButtonBackgroundDefault);
m_Button->SetPressedImage(ButtonBackgroundSelected);

m_TextBox->SetXYPos(Abs.x, Abs.y);
m_TextBox->SetWidthHeight(DropDownWidth-ButtonWidth, DropDownHeight);
m_TextBox->SetFocus(true);

return true;
}
```

### 12.6.3 Showing and Hiding the List

Drop-down lists can be in one of two states that refer to whether their list is shown or not; these are dropped down or not dropped down. Ultimately, this corresponds to whether the list box is visible or not. The following ShowDropDownList method of CXDropDownList has been coded to switch the list's visible status according to a passed Boolean argument. This method is invoked on OnMouseDown and OnMouseUp events. Specifically, this will change the list's visibility when the user clicks the drop-down list's button.

```
void CXDropDownList::ShowDropDownList(bool State)
{
    m_ListBox->SetVisible(State);
    IsDroppedDown = State;

    if(State)
        SetWidthHeight(GetWidth(), m_RolledUpHeight +
                        m_ListBox->GetHeight());
    else
        SetWidthHeight(GetWidth(), m_RolledUpHeight);
}
```

## 12.6.4 Handling Input

User input affects which mode the drop-down list is in. On a mouse click, an `OnMouseDown` event will be generated and this could lead to a number of different outcomes: One, the user may have clicked within the text box which implies he will input text; two, the user could have clicked on the button and therefore the list should be shown; three, if the list is shown, the user may click an item and thereby the list disappears and the item text overwrites that of the text box's; and finally, the list may be visible and the click may have occurred over the list's scroll bar, in which case the list should scroll and not disappear. The following two events, `OnMouseDown` and `OnKeyDown`, show a sample of how input is handled. On the companion CD, more events are handled.

```
void CXDropDownList::OnMouseDown(int Button, int X, int Y)
{
    if(TextBoxIntersect(X,Y))
    {
        m_TextBox->OnMouseDown(Button, X, Y);
        ShowDropDownList(false);
        return;
    }

    if(ButtonIntersect(X,Y))
    {
        m_Button->OnMouseDown(Button, X, Y);
        ShowDropDownList(true);
    }

    if(ListBoxIntersect(X,Y))
    {
        m_ListBox->OnMouseDown(Button, X, Y);

        if(ListItemIntersect(X,Y))
        {
            if(m_ListBox->GetSelected())
            {
                char* Text = (char*) m_ListBox->GetSelected()->GetCaption();
```

```

        m_TextBox->SetText(Text);
    }

    ShowDropDownList(false);
}

}

//-----

void CXDropDownList::OnKeyDown(WPARAM Key, LPARAM Extended)
{
    ShowDropDownList(false);
    m_TextBox->OnKeyDown(Key, Extended);
}

```

## 12.6.5 Painting

Painting a drop-down list is a particularly simple process. You simply need to initiate the `OnRender` events in the text box, list box, and button, and the rest all falls into place.

```

bool CXDropDownList::OnRender()
{
    D3DXVECTOR2 Abs;
    ZeroMemory(&Abs, sizeof(D3DXVECTOR2));
    GetAbsolutePosition(&Abs);

    m_TextBox->SetXYPos(Abs.x, Abs.y);
    m_TextBox->OnRender();

    m_Button->SetXYPos(Abs.x + (GetWidth()-m_Button->GetWidth()), Abs.y);
    m_Button->OnRender();

    m_ListBox->SetXYPos(Abs.x, Abs.y + m_RolledUpHeight);
    m_ListBox->OnRender();

    return true;
}

```

## 12.6.6 Cleaning Up

Lastly, the drop-down lists frees its members using the following destructor.

```
CXDropDownList::~CXDropDownList()
{
    if(m_ListBox)
        delete m_ListBox;

    if(m_TextBox)
        delete m_TextBox;

    if(m_Button)
        delete m_Button;
}
```

## 12.7 Conclusion

---

Well, ladies and gentlemen, UI LIB is now completed. Of course, it is by no means comprehensive. You could add a veritable cornucopia of controls including menus, toolbars, date and time controls, and even tree views if you're feeling particularly inspired. But what has been covered thus far will have demonstrated a variety of different techniques that it's almost as if a pattern exists among developing controls, and thus creating more will be that much simpler having now grasped the crucial understanding that governs an interface library. At this point I recommend going back over Part II to review the chapter conclusions and then move on to Part III, which follows. Here, UI LIB will be put to the test as we develop an interface for a DirectShow-powered media player.

# Introducing DirectShow

Welcome to the final part of this book, Part III. At this point, UI LIB has been completed and sports a number of different controls for interface development, namely labels, buttons, text boxes, check boxes, list boxes, and drop-down lists. During the course of Part III, these controls will be employed to develop an interface for a DirectShow media player. The development of that application begins in this chapter as we examine DirectShow generally and explore how it can be used to play media files. Specifically, this chapter explains the following:

- How to enable Visual C++ projects to use DirectShow
- How to create DirectShow applications using the filter graph
- How to play media files like MPG, AVI, and MP3
- How to receive events from a playing file

## 13.1 DirectShow — What Is It?

---

Tracing its lineage from ActiveMovie, Microsoft DirectShow is part of DirectX and is a COM-based architecture for streaming media in Windows. It can capture media like video from cameras or audio from microphones, and it can play media like videos and music from files such as MPG and MP3. Specifically, this book employs DirectShow only to play media files, as is relevant to a media player. For our purposes, therefore, DirectShow can be thought of as a programmable media player. Among the file formats and compressions it supports are WMA, WMV, ASF, MPEG, WAV, MP3, DV, AIFF, and QuickTime (version 2 and lower), among others. The following are the basic steps involved in creating a DirectShow application from beginning to end, using its interfaces to play media files. Subsequent sections examine these points in more detail.

### 1. **Include headers and libraries**

DirectShow applications should include `dshow.h` and link to `Strmiids.lib` and `Quartz.lib`.

### 2. **Initialize COM**

DirectShow applications must begin by initializing COM. This involves calling `CoInitialize` and then calling `CoUninitialize`.

### 3. **Create the filter graph**

Every DirectShow application must create a filter graph. This represents the mechanism by which a media file is decoded and played. Filter graphs are constructed by graph builders (`IGraphBuilder`), and this interface is created by calling the `CoCreateInstance` function.

### 4. **Create a media control**

After creating a filter graph representing a playable media file, you should create a media control to perform basic operations with it, such as playing the file or stopping the file. This is achieved by the `IMediaControl` interface and is also returned from `CoCreateInstance`.

**5. Create and configure an event mechanism**

You're going to want to know when file playback stops, when errors occur, and when the user cancels operations. This can be determined through the `IMediaEventEx` interface. Like other interfaces in DirectShow, this is created by the `CoCreateInstance` function.

**6. Load a file**

You should then construct a filter graph to play a specific file. This can be easily achieved by calling `IGraphBuilder::RenderFile`.

**7. Play the file**

Once a graph builder has constructed a filter graph to play a media file, and once an event interface has been instantiated, the media control interface can be used to play the file.

**8. Process media events**

As a media file plays, a number of events occur — the file may come to an end, the file may encounter an error, or the user may cancel playback. To notify you of this, a custom message is sent to `WndProc`, indicating that `IMediaEventEx` should be polled for information.

**9. Clean up**

When you are finished using DirectShow, you should clean up by releasing your instantiated interfaces and uninitializing COM.

## 13.2 Getting Started

So how do you create a DirectShow application? Simple! You can change your Direct3D projects to include the relevant libraries and header or create a new program from scratch. Figure 13.1 demonstrates how the linker options should appear.

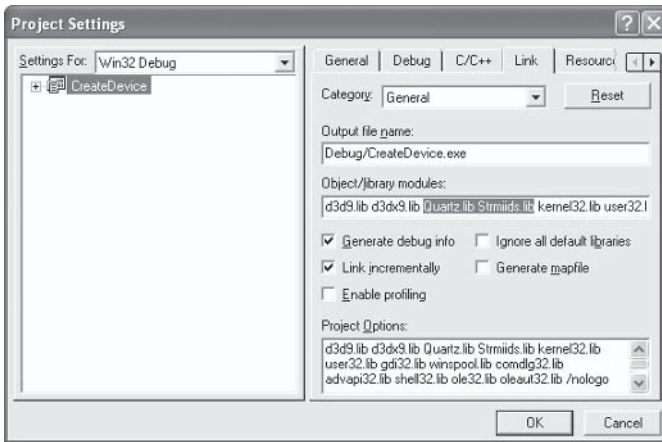


Figure 13.1

To ensure DirectShow can compile and run, you must include `dshow.h` and link to `Strmiids.lib` and `Quartz.lib`. You must also ensure that you link to `ole32.lib` and `oleaut32.lib` to ensure the standard COM functions, such as `CoCreateInstance`, can compile. Finally, you must ensure your application initializes the COM library in some initialization procedure. This involves a call to `CoInitialize`. For clarity, this process and more will occur in a sample procedure called `InitDirectShow`, as shown below. It contains all the DirectShow initialization code for a sample application. The next section examines the first stages of programming DirectShow.

```
HRESULT InitDirectShow (HWND hWnd)
{
    HRESULT hr = CoInitialize(NULL);
    if (SUCCEEDED(hr))
    {
    }
    return hr;
}
```

## 13.3 The Filter Graph

---

DirectShow plays media files using the filter graph. This is basically a conceptual diagram that opens, decodes, and plays a file. It's composed from a series of process blocks, or modules, that are connected to one another by pins. Media data enters the graph at a starting point, and then travels along the pins and into each module. Modules take in information, process it, and then output a result. Thus, data moves along the graph, passing through module after module until it exits in the form of something playable. Fortunately, developers don't often need to be concerned about the intricacies of a filter graph's components since DirectShow can automatically assemble them to play common media files. However, developers do need to create the `IGraphBuilder` interface to use DirectShow, and it is with this interface that DirectShow assembles a filter graph. This interface can be created by calling the `CoCreateInstance` function. Its declaration and parameters follow, and after that a continuation of `InitDirectShow` demonstrates how to create a filter graph builder.

```

STDAPI CoCreateInstance(
    REFCLSID rclsid,
    LPUNKNOWN pUnkOuter,
    DWORD dwClsContext,
    REFIID riid,
    LPVOID *ppv
);

```

---

**REFCLSID *rclsid*:** CLSID representing the class associated with the object to create. For DirectShow graph builders, this value should be CLSID\_FilterGraph.

---

**LPUNKNOWN *pUnkOuter*:** Don't worry about this. Just pass NULL. For more information, please consult the Windows SDK.

---

**DWORD *dwClsContext*:** Context in which the returned object will run. For standard DirectShow applications, this value should be CLSCTX\_INPROC\_SERVER.

---

**REFIID *riid*:** Identifier of the interface to be retrieved. For the DirectShow filter graph builder, this should be IID\_IGraphBuilder.

---

**LPVOID *\*ppv*:** Address where the specified interface is to be returned.

---

```

HRESULT InitDirectShow (HWND hWnd)
{
    HRESULT hr = CoInitialize(NULL);
    if (SUCCEEDED(hr))
    {
        hr = CoCreateInstance(CLSID_FilterGraph, NULL,
            CLSCTX_INPROC_SERVER, IID_IGraphBuilder, (void **)&pGraph);
    }
    return hr;
}

```

## 13.4 The Media Control

Once IGraphBuilder has been created, an application then has created a blank filter graph. This is now a blank canvas that is set to be built in order to play a media file, such as a movie or a song. Before populating the graph with all the blocks, pins, and connectors to play such a file, you should first create a media control. This will act like a remote control that will allow you to stop and start playback of a media file running through the filter graph. The media control is represented by the IMediaControl interface and is returned by calling IGraphBuilder's QueryInterface method. InitDirectShow can be appended to include this.

```
HRESULT InitDirectShow (HWND hWnd)
{
    HRESULT hr = CoInitialize(NULL);
    if (SUCCEEDED(hr))
    {
        hr = CoCreateInstance(CLSID_FilterGraph, NULL,
            CLSCTX_INPROC_SERVER, IID_IGraphBuilder, (void **)&pGraph);

        hr = pGraph->QueryInterface(IID_IMediaControl, (void **)&pControl);
    }
    return hr;
}
```

## 13.5 The Event Mechanism

One other step an application should perform before loading a media file and building the filter graph is creating an event interface. This is explained in more detail later. Its purpose is to assist an application in determining when a file has stopped playing or when a large number of other events occur while a media file is playing. This is represented by the IMediaEventEx interface and can be returned by IGraphBuilder's QueryInterface method.

```
HRESULT InitDirectShow (HWND hWnd)
{
    HRESULT hr = CoInitialize(NULL);
    if (SUCCEEDED(hr))
    {
        hr = CoCreateInstance(CLSID_FilterGraph, NULL,
            CLSCTX_INPROC_SERVER, IID_IGraphBuilder, (void **)&pGraph);

        hr = pGraph->QueryInterface(IID_IMediaControl, (void **)&pControl);
        hr = pGraph->QueryInterface(IID_IMediaEventEx, (void **)&g_pEvent);
    }
    return hr;
}
```

## 13.6 Registering for Events

---

IMediaEventEx notifies applications as events occur during media playback. It does this generally by sending a custom Windows message to MsgProc, telling an application that some event has happened. Applications should then handle the message by querying the IMediaEventEx interface to find out specifically what happened. The process of handling this message and events is discussed later. However, during initialization, an application must first notify IMediaEventEx about which window a message should be sent to and which message it is expected to send. This is called registering for events. To do this, an application must first define its own message, such as:

```
#define WM_GRAPHNOTIFY WM_APP + 1
```

Then, you register for events using the SetNotifyWindow method of IMediaEventEx. The declaration and parameters for this function are listed below, and then InitDirectShow has been coded to perform the registering process.

```

HRESULT SetNotifyWindow(
    OAHWND hwnd,
    long lMsg,
    long lInstanceData
);
    
```

---

**OAHWND *hwnd***: Handle of the window to receive notifications or NULL to stop receiving messages.

---

**long *lMsg***: Window message you want to use as a notification. For the purposes of this example, this value should be WM\_GRAPHNOTIFY.

---

**long *lInstanceData***: Value that is to be used for the LPARAM of the notification.

---

```

HRESULT InitDirectShow (HWND hWnd)
{
    HRESULT hr = CoInitialize(NULL);
    if (SUCCEEDED(hr))
    {
        hr = CoCreateInstance(CLSID_FilterGraph, NULL,
            CLSCTX_INPROC_SERVER, IID_IGraphBuilder, (void **)&pGraph);

        hr = pGraph->QueryInterface(IID_IMediaControl, (void **)&pControl);
        pGraph->QueryInterface(IID_IMediaEventEx, (void **)&g_pEvent);
        g_pEvent->SetNotifyWindow((OAHWND)hWnd, WM_GRAPHNOTIFY, 0);
    }
    return hr;
}
    
```

## 13.7 Loading a File

---

Once an application has created IGraphBuilder, IMediaControl, and IMediaEventEx, a file can be loaded into DirectShow for playback via the filter graph. This is achieved simply by calling the RenderFile method of IGraphBuilder. Calling this method loads a specified file, ready for playback, and automatically builds a filter graph to accommodate it. Note, however, that this function does not actually

play a file. This is performed in the next section. Meanwhile, the function declaration and parameters for `RenderFile` follow, and beneath that continues the definition of `InitDirectShow`, which now demonstrates how to load a sample MP3 file.

```
HRESULT RenderFile(  
    LPCWSTR lpwstrFile,  
    LPCWSTR lpwstrPlayList  
);
```

---

**LPCWSTR *lpwstrFile***: A wide char specifying the name of the media file. This can be any media from MP3 to MPG to AVI. It could also be a local file, such as those on your hard drive, or even an online resource, like a file on the web.

---

**LPCWSTR *lpwstrPlayList***: Just pass NULL.

---

```
HRESULT InitDirectShow (HWND hWnd)  
{  
    HRESULT hr = CoInitialize(NULL);  
    if (SUCCEEDED(hr))  
    {  
        hr = CoCreateInstance(CLSID_FilterGraph, NULL,  
            CLSCTX_INPROC_SERVER, IID_IGraphBuilder, (void **)&pGraph);  
  
        hr = pGraph->QueryInterface(IID_IMediaControl, (void **)&pControl);  
        pGraph->QueryInterface(IID_IMediaEventEx, (void **)&g_pEvent);  
        g_pEvent->SetNotifyWindow((OAHWND)hWnd, WM_GRAPHNOTIFY, 0);  
        hr = pGraph->RenderFile(L"sample.mp3", NULL);  
    }  
    return hr;  
}
```

## 13.8 Playing a File

Playing a loaded file is one of the simplest processes to perform. It involves calling the Run method of IMediaControl. It takes no parameters and immediately begins playback via the filter graph. The Run method can be called anywhere in an application to begin playback. For the sake of clarity, I have done this inside the InitDirectShow procedure. This can be coded as follows.

```
HRESULT InitDirectShow (HWND hWnd)
{
    HRESULT hr = CoInitialize(NULL);
    if (SUCCEEDED(hr))
    {
        hr = CoCreateInstance(CLSID_FilterGraph, NULL,
            CLSCTX_INPROC_SERVER, IID_IGraphBuilder, (void **)&pGraph);

        hr = pGraph->QueryInterface(IID_IMediaControl, (void **)&pControl);
        pGraph->QueryInterface(IID_IMediaEventEx, (void **)&g_pEvent);
        g_pEvent->SetNotifyWindow((OAHWND)hWnd, WM_GRAPHNOTIFY, 0);
        hr = pGraph->RenderFile(L"sample.mp3", NULL);

        hr = pControl->Run();
    }
    return hr;
}
```

## 13.9 Catching Media Events

Once a media file is playing, it's likely to trigger a number of events. These events might be the end of a song or a user canceling playback. It notifies applications of these events by sending a custom Windows message to MsgProc, telling an application that some event has happened. This message is the same one that was passed to IMediaEventEx in Section 13.6. When an application receives this, it knows that an event occurred and that it should

query `IMediaEventEx` for more information. The process of doing this is handled in the next section. The following code, however, demonstrates a sample `WndProc` procedure that captures this message and passes it on to an application-defined function called `HandleGraphEvent`. This function is used in the next section to illustrate how messages are processed by `IMediaEventEx`.

```
LRESULT WINAPI MsgProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
    switch(msg)
    {
        case WM_DESTROY:
            Cleanup();
            PostQuitMessage(0);
            return 0;

        case WM_PAINT:
            ValidateRect(hWnd, NULL);
            return 0;

        case WM_GRAPHNOTIFY:
            HandleGraphEvent();
            return 0;
    }

    return DefWindowProc(hWnd, msg, wParam, lParam);
}
```

## 13.10 Reading Media Events

---

Once an application receives a custom message in `WndProc`, it knows one or more events have occurred during media playback and may need to be handled. For example, if a song has finished playing, you might want to load another song for playback, especially if the user has made some kind of playlist. Thus, you need to query `IMediaEventEx` to see what event or events occurred and process them or ignore them accordingly. You read events from `IMediaEventEx` by calling `GetEvent`. Its declaration and parameters

follow, followed by Table 13.1, which lists common events that could be returned. The next section handles some of these events.

```
HRESULT GetEvent(
    long *lEventCode,
    LONG_PTR *lParam1,
    LONG_PTR *lParam2,
    long msTimeout
);
```

---

**long \*lEventCode:** Address of a variable to receive the event code.

---

**LONG\_PTR \*lParam1:** Address of a variable to receive the first event parameter.

---

**LONG\_PTR \*lParam2:** Address of a variable to receive the second event parameter.

---

**long msTimeout:** You can specify an interval of time to suspend program execution and wait for the next event. Use INFINITE to block until an event occurs. Since our application receives notifications through MsgProc, there's no need to wait, so just pass 0.

---

**Table 13.1. Common events**

<p><b>EC_ACTIVATE</b></p>	<p>A separate window playing a video has been activated or deactivated.</p> <p><b>lParam1</b> True means the window was activated and false means it was deactivated.</p> <p><b>lParam2</b> Pointer to an IBaseFilter interface.</p>
<p><b>EC_BUFFERING_DATA</b></p>	<p>The filter graph is starting to buffer or has stopped buffering data.</p> <p><b>lParam1</b> If data is starting to buffer, this value is true; else it is false.</p> <p><b>lParam2</b> This value is 0.</p>

<b>EC_COMPLETE</b>	<p>This event occurs when media has successfully finished playing.</p> <p><b>IParam1</b> HRESULT value.</p> <p><b>IParam2</b> This value is 0 or a pointer to IBaseFilter.</p>
<b>EC_ERRORABORT</b>	<p>Indicates that playback was aborted due to an error.</p> <p><b>IParam1</b> HRESULT value of the failed operation.</p> <p><b>IParam2</b> This value is 0.</p>
<b>EC_REPAINT</b>	<p>Video requires repainting.</p> <p><b>IParam1</b> Pointer to a PIN interface or NULL.</p> <p><b>IParam2</b> This value is 0.</p>
<b>EC_USERABORT</b>	<p>Occurs when the user cancels playback.</p> <p><b>IParam1</b> This value is 0.</p> <p><b>IParam2</b> This value is 0.</p>

**TIP**

For a list of notification codes and descriptions, please see the DirectX SDK.

## 13.11 Handling Media Events

After a custom message is received by `WndProc`, an application knows that an event occurred during media playback. It should then handle this message by checking `IMediaEventEx` for pending events using the `GetEvent` function, as demonstrated in the previous section. However, one important thing to remember is that there may be more than one event waiting to be processed and `GetEvent` only returns one event at a time. Therefore, you should continually call this function until it fails, as each time it will return information about a different event. If the method succeeds, the relevant information is returned at the specified addresses. If the method fails, there are no more events to process. The following code from an application-defined function, `HandleGraphEvent`, demonstrates how messages can be retrieved from `IMediaEventEx`.

```
void HandleGraphEvent()
{
    if(g_pEvent)
    {
        long evCode;
        LONG_PTR param1, param2;

        while (SUCCEEDED(g_pEvent->GetEvent(&evCode, &param1, &param2, 0)))
        {
            switch (evCode)
            {
                case EC_COMPLETE:
                case EC_USERABORT:
                case EC_ERRORABORT:
                    return;
            }
            g_pEvent->FreeEventParams(evCode, param1, param2);
        }
    }
}
```

## 13.12 Cleaning Up

---

Finally, once you've finished using DirectShow, you should then free its instantiated interfaces and uninitialized COM. This has been performed in the following application-defined Cleanup function.

```
VOID Cleanup()
{
    if(pGraph)
        pGraph->Release();

    if(pControl)
        pControl->Release();

    if(g_pEvent)
        g_pEvent->Release();

    CoUninitialize();
}
```

## 13.13 Conclusion

---

This chapter has presented a brief overview of the awesome power of DirectShow and provided us with a good basis upon which to build a media player application. The next chapter concentrates on wrapping DirectShow into a form that is particularly favorable to media players. Before moving on, however, let's recap the key points explained in this chapter.

- DirectShow plays media files using the filter graph, which is a conceptual diagram that opens, decodes, and plays a file. This is constructed by IGraphBuilder.
- IMediaControl is an interface that controls the filter graph's playback.
- IMediaEventEx is used to notify applications of media events that occur.

# Wrapping DirectShow

This chapter continues from where the previous ended by encapsulating DirectShow into two wrapper classes. These are geared toward creating a media player in the following chapter. Specifically, this chapter demonstrates the following:

- How to encapsulate media files into class CXMedia
- How to create a list of media classes to form a playlist
- How to encapsulate IGraphBuilder, IMediaControl, and IMediaEventEx
- How to use this class to play a playlist

## 14.1 CXMedia and CXMediaPlayer

---

DirectShow contains much functionality, much of which is irrelevant to this book. However, its ability to play media files will be encapsulated over the following two classes. Subsequent sections examine their implementation.

The CXMedia class encapsulates a media file that is to be played by the CXMediaPlayer class. This equates to storing a valid path to a media file. It also will be linked list enabled by storing a next pointer, which makes it easy to maintain a playlist of media.

The `CXMediaPlayer` class is the real meat and potatoes of the operation. This class encapsulates the `IGraphBuilder`, `IMediaControl`, and `IMediaEventEx` interfaces. These will be used to play media files — or more specifically, will be used to play one or more instances of `CXMedia`. These will be stored in a linked list and played sequentially.

## 14.2 **CXMedia — Songs, Movies, and More**

---

Media for media players consists mainly of movies and music, and these are divided into different file formats, namely, MPG, MPEG, AVI, WAV, MP3, SND, etc. All types of media like this can be represented in class `CXMedia`. Its entire declaration and definition follows. This class stores several properties about a single media file, including its file path, whether the file is currently being played, and how many times it should be repeated, if any. Take a look at its code below to get a taste of how it works. The next section examines how this is played by class `CXMediaPlayer`.

```
class CXMedia
{
private:
protected:
    std::string m_Media;
    int m_Repeats;
    CXMedia* m_Next;
    bool m_IsPlaying;
public:
    CXMedia();
    void SetMedia(const char* Path) {m_Media = Path;}
    const char* GetMedia() {return m_Media.c_str();}
    void SetRepeats(int Repeats) {m_Repeats = Repeats;}
    int GetRepeats() const {return m_Repeats;}
    CXMedia* GetNext() const {return m_Next;}
    void SetNext(CXMedia* Next) {m_Next = Next;}
    void SetPlaying(bool State) {m_IsPlaying = State;}
    bool GetPlaying() const {return m_IsPlaying;}
```

```
};

CXMedia::CXMedia()
{
    m_Repeats = 0;
    m_Next = NULL;
    m_IsPlaying = false;
}
```

## 14.3 CXMediaPlayer — Player of the Playlist

---

Media files are represented by class `CXMedia`. This defines the path to a media file and the number of times a player should repeat it. Instances of `CXMedia` are intended to be played by `CXMediaPlayer`. This encapsulates `IGraphBuilder`, `IMediaControl`, and `IMediaEventEx` and performs the usual DirectShow tedium that was demonstrated in the previous chapter. The code below presents the class declaration and the key points of this class follow.

```
class CXMediaPlayer
{
private:
protected:
    CXMedia* m_PlayList;
    CXMedia* m_CurrentlyPlaying;
    IGraphBuilder *m_Graph;
    IMediaControl *m_Control;
    IMediaEventEx *m_pEvent;
    HWND m_hwnd;
    int m_Counter;
    void Initialize(); //Initializes DirectShow objects
    void Free(); //Clears DirectShow objects
public:
    CXMediaPlayer(HWND hwnd);
    ~CXMediaPlayer();
    void ClearPlaylist();
    CXMedia* AddMediaToPlaylist(const char* Path, int Repeats);
```

```

void Play(CXMedia* Media);
void Stop();
void Pause();
void Update();
CXMedia* GetFirstItem() {return m_PlayList;}
CXMedia* GetPlayingItem() {return m_CurrentlyPlaying;}
};

```

- Class `CXMediaPlayer` plays media files of type `CXMedia`. It stores them as a playlist and maintains a pointer to the first item (`m_PlayList`).
- `CXMediaPlayer` plays media for its specified number of repeats through the `Play` method and keeps track of the currently playing item through `m_CurrentlyPlaying`.
- As playback finishes, `CXMediaPlayer` increments a counter of how many plays the current item has had (`m_Counter`). This is compared to the item's repeat count to determine whether the next item in the playlist should be played. A repeat count of `-1` means it should repeat indefinitely.

### 14.3.1 The Class Constructor

Applications that declare instances of the media player implicitly call its constructor. This sets all of its member variables to starting values and initializes COM, ready to use `DirectShow`. Its definition follows.

```

CXMediaPlayer::CXMediaPlayer(HWND hWnd)
{
    m_Graph = NULL;
    m_Control = NULL;
    m_pEvent = NULL;
    m_PlayList = NULL;
    m_CurrentlyPlaying = NULL;
    m_Counter = 0;
    m_hWnd = hWnd;

    CoInitialize(NULL);
}

```

 **TIP**

You may want to call `Colnitialize` at application start, especially if you decide to change COM mode or use other COM interfaces.

## 14.3.2 Initializing DirectShow

Once a media player object is created, one of the first things that should be done is the initialization of DirectShow, as demonstrated in the previous chapter. Briefly, this involves creating the filter graph through which media travels, creating the media control to start, stop, and pause playback, and creating the event mechanism by which the media player can be notified of events during media playback. Actually, this process is performed implicitly as the class is called upon to play media files. Initialization is achieved with the `Initialize` method, and it can be coded to initialize DirectShow as follows.

```
void CXMediaPlayer::Initialize()
{
    Free(); //Deletes any previous DirectShow instances

    CoCreateInstance(CLSID_FilterGraph, NULL,
        CLSCTX_INPROC_SERVER, IID_IGraphBuilder, (void **)&m_Graph);

    m_Graph->QueryInterface(IID_IMediaControl, (void **)&m_Control);
    m_Graph->QueryInterface(IID_IMediaEventEx, (void **)&m_pEvent);
    m_pEvent->SetNotifyWindow((OAHWND)m_hWnd, WM_GRAPHNOTIFY, 0);
}
```

## 14.3.3 Adding Media Files

You might have guessed by now that I like using linked lists. This is because they — or adaptations of them — are useful for a whole range of scenarios, and storing a playlist is just one of them. By definition, a playlist refers to a list of media files that are scheduled to be played sequentially, one after the other. The user doesn't need to prompt a media player for this to occur. The user can simply load in his playlist and away it goes. `CXMediaPlayer` stores a playlist as

an array of `CXMedia` classes — or more specifically, as a linked list. This means the list can grow and shrink dynamically, as media is added or removed. The process of adding files is simple. This can be seen below using the following `AddMediaToPlayList` method of `CXMediaPlayer`.

```
CXMedia* CXMediaPlayer::AddMediaToPlayList(const char* Path, int Repeats)
{

    CXMedia* Media = new CXMedia();
    Media->SetMedia(Path);
    Media->SetRepeats(Repeats);

    if(!m_PlayList)
        m_PlayList = Media;
    else
    {
        CXMedia* Temp = m_PlayList;

        while(Temp->GetNext())
            Temp = Temp->GetNext();

        Temp->SetNext(Media);
    }

    return Media;
}
```

### 14.3.4 Clearing Media Files

There are going to be times when you'll want to clear the playlist, perhaps because you're loading a new set of media or deleting the media player from memory. The following `ClearPlayList` method has been added to `CXMediaPlayer` to achieve this simple task, a process that we have seen before in other contexts.

```
void CXMediaPlayer::ClearPlayList()
{
    CXMedia* Temp = m_PlayList;

    while(Temp)
    {
        CXMedia* Next = Temp->GetNext();

        if(Temp)
            delete Temp;

        Temp = Next;
    }

    m_PlayList = NULL;
}
```

### 14.3.5 Playing a Playlist

Once one or more media files have been added to the playlist, the media player class can then play items. It does this through the `Play` method, which requires a starting item in the playlist where playback is to begin. For example, if you wanted to play the whole list, from beginning to end, then you'd simply pass the first media item as the `Play` method's argument.

```
void CXMediaPlayer::Play(CXMedia* Media)
{
    if(Media)
    {
        if(m_CurrentlyPlaying) //Resume paused file
        {
            m_Control->Run();
        }
        else //Play new file
        {
            Initialize();
            const char* Str = Media->GetMedia();
            int Length = strlen(Str);
```

```
WCHAR* Path = new wchar_t [Length + 1];

MultiByteToWideChar(CP_ACP, 0, Str, Length+1, Path, Length + 1);

m_Control->Stop();
m_Graph->RenderFile(Path, NULL);
m_Control->Run();

m_CurrentlyPlaying = Media;
Media->SetPlaying(true);

return;
}
}
m_CurrentlyPlaying = NULL;
}
```

**NOTE**

Notice that `RenderFile` requires its media source to be specified as a wide string, rather than the standard char. Thus, the `MultiByteToWideChar` function has been used to convert the returned string from `std::string`'s `c_str` method.

## 14.3.6 Pausing and Stopping

Apart from playing media files, it is also useful to stop and pause them, a subject not addressed in the previous chapter. However, it doesn't require much explanation since `IMediaControl` supports both `Stop` and `Pause` methods and neither requires arguments. The following `Stop` and `Pause` methods of `CXMediaPlayer` encapsulate them, and their definition can be coded as follows.

```
void CXMediaPlayer::Stop()
{
    m_Counter = 0;
    m_CurrentlyPlaying = NULL;
    if(m_Control)
        m_Control->Stop();
}
```

```
void CXMediaPlayer::Pause()
{
    if(m_Control)
        m_Control->Pause();
}
```

## 14.3.7 Handling Messages

Once a media file is playing, CXMediaPlayer must know when events occur. Specifically, it must know when playback is completed in order to kick off the next file playing, stop playback, or repeat the current item. To notify CXMediaPlayer that an event occurred, an application must watch for the custom message in WndProc and then invoke CXMediaPlayer's Update method where events are handled. Take a look at its function definition below to see how it handles EC\_COMPLETE.

```
void CXMediaPlayer::Update()
{
    if(m_pEvent)
    {
        long evCode;
        LONG_PTR param1, param2;

        while (SUCCEEDED(m_pEvent->GetEvent(&evCode, &param1, &param2, 0)))
        {
            switch (evCode)
            {
                case EC_COMPLETE:

                    if(m_CurrentlyPlaying)
                    {
                        if((m_CurrentlyPlaying->GetRepeats() == -1) ||
                            (m_Counter < m_CurrentlyPlaying->GetRepeats()))
                        {
                            m_Counter++;
                            Play(m_CurrentlyPlaying);
                        }
                        else
                        {

```

```
        m_CurrentlyPlaying->SetPlaying(false);
        CXMedia* Next = m_CurrentlyPlaying->GetNext();
        m_CurrentlyPlaying = NULL;
        m_Counter = 0;
        Play(Next);
    }
}
break;
}
m_pEvent->FreeEventParams(evCode, param1, param2);
}
}
}
```

### 14.3.8 Uninitializing DirectShow

Once you've finished using DirectShow, you should uninitialize its interfaces by calling their release methods. Usually, this will occur in the class destructor, but it has been coded as a separate method that will always be invoked there. This method is called `Free` and its definition is given below.

```
void CXMediaPlayer::Free()
{
    if(m_Graph)
        m_Graph->Release();

    if(m_Control)
        m_Control->Release();

    if(m_pEvent)
        m_pEvent->Release();
}
```

### 14.3.9 Cleaning Up

Finally, once an application has finished using the media player, it should delete it from memory. This invokes its class destructor where its members are consequently released.

```
CXMediaPlayer::~CXMediaPlayer()
{
    Free();
    ClearPlayList();
    CoUninitialize();
}
```

## 14.4 Conclusion

---

This chapter has presented an overview of how to encapsulate DirectShow. It supports playlists, and media can be played, paused, and stopped. One potential improvement that could be made is the inclusion of media seeking such that a user can jump backward and forward to specific points in a media file. This is achieved by the IMediaSeeking interface but is beyond the scope of this book. To learn more about this and DirectShow in general, please consult the appendix. However, the classes thus far developed facilitate our needs adequately and it's now time for us to move on. The following chapter is not only the last, but also demonstrates how our foundational knowledge from Part I, UI LIB from Part II, and our media player work during Part III come together to form a media player application, complete with an interface.

This page intentionally left blank.

---

# Building the Media Player

---

Congratulations! You've reached the final chapter. Here, we'll create a media player application, complete with an interface. To do this, we'll be using the DirectShow wrappers from the previous chapter and an assortment of controls from UI LIB, developed in Part II. This chapter demonstrates the following:

- How to use CXMedia and CXMediaPlayer
- How to use buttons, labels, text boxes, list boxes, scroll bars, and windows
- How to override buttons to handle clicks differently
- How to implement connectivity between controls

## 15.1 The Media Player

---

As mentioned, this chapter focuses on creating a basic media player application. Primarily, this application is geared toward playing a playlist of MP3 music files, but it can also play video and other media. Specifically, it allows the user to enter a search path. Having done so the user can click the search button, whereupon a list of MP3s found at that destination are loaded into the playlist. These

can be played, stopped, or paused using the media player buttons. See Figure 15.1 for a glimpse at the final program.



Figure 15.1

## 15.2 CXMyMediaPlayerApp — The Media Player

The entire media player application is coded into a single class. Hence, this means that any Visual C++ project wanting to become a media player can just instantly plug in our class. Specifically, this class will be derived from `CXWindow` and can be set to act as a parent window, or desktop. Recall from Chapter 9 that `CXWindow` can encapsulate the largest frame inside which an interface exists, including other windows and controls. To this extent therefore, our class will be the parent window in an MDI application, or like the Windows desktop; it is a container for an interface. Therefore, applications essentially become media players by instantiating this

class to be its desktop, or topmost window. The class will be named `CXMyMediaPlayerApp` and its declaration and key points follow.

```
class CXMyMediaPlayerApp : public CXDesktopWindow
{
private:
protected:
    CXWindow* m_Window;
    CXListBox* m_FileList;
    CXPlayButton* m_PlayButton;
    CXStopButton* m_StopButton;
    CXSearchButton* m_SearchButton;
    CXCloseButton* m_CloseButton;
    CXMediaPlayer* m_MediaPlayer;
    CXTextBox* m_SearchPath;
    void LoadWindows();
    void LoadListBoxes();
    void LoadButtons();
    void LoadTextBoxes();
public:
    CXMyMediaPlayerApp(HWND hWnd, LPDIRECT3DDEVICE9 Device);
    ~CXMyMediaPlayerApp();
    bool Intialize(int Width, int Height);
    bool OnRender();
    void OnMouseDown(int Button, int X, int Y);
    void OnMouseMove(int X, int Y);
    void OnMouseUp(int Button, int X, int Y);
    void OnSetFocus();
    void OnLostFocus();
    void OnKeyDown(WPARAM Key, LPARAM Extended);
    void OnKeyUp(WPARAM Key, LPARAM Extended);
    void OnSysKey(WPARAM Key, LPARAM Extended);
    void Update() {m_MediaPlayer->Update();}
};
```

- Class `CXMyMediaPlayerApp` allows an application to become a media player. It implements both the front end and back end of a media player application. This means it implements both the interface and inner workings.

- Class `CXMyMediaPlayerApp` represents a desktop, or parent window, that contains an interface for a media player. Specifically, it contains a single child window (`m_Window`) and its child controls, which are used to present a media player and control its operations.
- `CXMyMediaPlayerApp` maintains a media player and playlist through member `m_MediaPlayer`. This class was developed in the previous chapter and is used to play MP3 files.
- `m_Window` has a number of child controls that are used to present the application. These are lists, text boxes, and customized buttons that perform different operations. The controls this window contains are as follows:
  - **`CXListBox* m_FileList`** — List box representing the playlist the media player should play.
  - **`CXPlayButton* m_PlayButton`** — Button that switches between play and paused. Clicking on this for the first time plays the playlist. Clicking again pauses playback, and clicking yet again resumes playback.
  - **`CXStopButton* m_StopButton`** — This button stops playback of the current playlist.
  - **`CXTextBox* m_SearchPath`** — Text box where a user enters a path to search for MP3 files.
  - **`CXSearchButton* m_SearchButton`** — This button creates a playlist by searching a specified path, defined by `m_SearchPath`, for MP3 files. The found files are then loaded into the list box `m_FileList` for playback.
  - **`CXCloseButton* m_CloseButton`** — This button closes the application.

## 15.2.1 Creating the Media Player

As mentioned, `CXMyMediaPlayerApp` represents a media player application, both interface and back end included. As instances of this class are instantiated, the class constructor is called to initialize its members to starting values. Inside this function, the media player class, `CXMediaPlayer`, is created. However, interface controls, such as buttons and lists, are created and initialized inside the `Initialize` method, which is handled in the next section onward. The definition below demonstrates how the constructor works.

```
CXMyMediaPlayerApp::CXMyMediaPlayerApp(HWND hWnd, LPDIRECT3DDEVICE9 Device)
    : CXDesktopWindow(Device)
{
    m_MediaPlayer = new CXMediaPlayer(hWnd);
    m_Window = NULL;
    m_FileList = NULL;
    m_PlayButton = NULL;
    m_StopButton = NULL;
    m_SearchButton = NULL;
    m_CloseButton = NULL;
    m_SearchPath = NULL;
}
```

## 15.2.2 Loading Controls

The interface for our media player program consists of the controls listed in Section 15.2. These are contained inside a window and include a list box, several buttons, and a text box. They allow the user to search a path for MP3 files, load them into a playlist, and then start, stop, or pause playback. These controls are loaded in groups by separate methods of `CXMyMediaPlayerApp`. Specifically, these methods are `LoadWindows`, `LoadListBoxes`, `LoadButtons`, and `LoadTextBoxes`, and they are all called by the `Initialize` method, which is used to load the media player's entire interface and to set the desktop frame's overall width and height. The definition for `Initialize` appears below, and the loading and development of individual controls is handled separately in subsequent sections.

```
bool CXMyMediaPlayerApp::Intialize(int Width, int Height)
{
    SetWidthHeight(Width, Height); //Set desktop dimensions
    LoadWindows(); //Load the window
    LoadListBoxes(); //Load the playlist list box
    LoadTextBoxes(); //Load the edit box to enter a search path
    LoadButtons(); //Load all buttons: play, stop, pause, search,
                  //and close

    return true;
}
```

### 15.2.3 The Window

The first control to examine is the main media player window. Remember that class `CXMyMediaPlayerApp` is derived from `CXWindow` and is therefore a window itself. However, this is not a visible window and simply acts as a frame, or desktop, encompassing the entire interface. Thus, another window must be created and should be made the child window of the desktop. It should be the standard type, `CXWindow`, as developed in Chapter 9, and should be visible, able to be dragged, and contain the controls described in Section 15.2. Figure 15.2 demonstrates the image I created in Photoshop to be the background of this window, and the following code shows the `LoadWindows` method of `CXMyMediaPlayerApp`. This instantiates an instance of `CXWindow` and then sets its starting properties. It also adds it to the control hierarchy as a child of the desktop.



Figure 15.2

```
void CXMyMediaPlayerApp::LoadWindows()  
{  
    m_Window = new CXWindow(GetDevice());  
    m_Window->LoadCanvasFromFile("Window_background.bmp");  
    m_Window->SetXYPos(0,0);  
    AddChildControl((CXControl*) m_Window);  
}
```



#### TIP

Of course, it's usually not a great idea to hard-code paths, as has been used for `LoadCanvasFromFile`. Typically, you would load this data from a file.

## 15.2.4 The List Box

Once the window (`m_Window`) is created, controls can be added to it. The first control, for no particular reason, will be the list box. This is going to be used to show a list of MP3 files at a specified path, and these will also be added to the playlist. Later, as the play button is pressed, the list is played sequentially, from top to bottom. The following images in Figures 15.3 to 15.8 are used for the list items' selected and un-selected states, and for the scroll bar thumb, background, and arrow buttons.



Figure 15.3. List item selected



Figure 15.4. List item unselected



Figure 15.5. Scroll bar thumb



Figure 15.6. Scroll bar background



Figure 15.7. Scroll bar up arrow



Figure 15.8. Scroll bar down arrow

The following `LoadListBoxes` function initializes the list box and adds it to the control hierarchy as a child of `m_Window`.

```
void CXMyMediaPlayerApp::LoadListBoxes()
{
    const FLOAT List_Item_Height = 32;

    m_FileList = new CXListBox(GetDevice());
    m_Window->AddChildControl((CXControl*) m_FileList);

    m_FileList->SetXYPos(6,75);
    m_FileList->SetWidthHeight(500, List_Item_Height*7);
}
```

```

m_FileList->LoadScrollBar("Background.jpg", "thumb.jpg", "up.jpg",
                          "down.jpg");
m_FileList->LoadItemBackgrounds("thumb.jpg", "thumb_Sel.jpg");
}
    
```

## 15.2.5 The Text Box

Another control to be added to and positioned inside `m_Window` is text box `m_SearchPath`. This allows a user to enter a search path to be searched for MP3 files. The following `LoadTextBoxes` function loads this control and inserts it into the control hierarchy as a child of the window `m_Window`.

```

void CXMyMediaPlayerApp::LoadTextBoxes()
{
    LOGFONT lf;
    SystemParametersInfo(SPI_GETICONTITLELOGFONT, sizeof(lf), &lf, 0);
    m_SearchPath = new CXTexBox(lf, GetDevice());
    m_SearchPath->SetXYPos(100,450);
    m_SearchPath->SetWidthHeight(300,32);
    m_SearchPath->SetText("c:\\");
    m_Window->AddChildControl((CXControl*) m_SearchPath);
}
    
```

## 15.3 Buttons Are Connectivity

Until now, controls have been added to member `m_Window` as relatively autonomous components. They have little interaction with one another at all. Thus, if the user enters a path into the text box to search for MP3 files, there's no implicit way the list box can know about it. Consequently, it cannot be populated with search results. Therefore, there needs to be a way in which some kind of connectivity can be established between controls such that one control can appropriately interact with another. This is where buttons come to the rescue. Buttons allow us to store pointers to other controls in the window and communicate between them. You may remember that buttons were developed in Chapter 10, as class

CXButton. Of course, this class was fine for sample applications but there was a small problem: Because buttons are all going to do different things, you need to handle mouse clicks differently. For example, a search button might search for a group of files, while a close button might end an application. Using CXButton exclusively doesn't allow you to do this. So, what should you do instead? Simple. You should add a virtual function — say `OnClick` — to CXButton and trigger it during `OnMouseDown` events. Then you simply override this event in a derived class to customize `OnClick` functionality. The following sections describe each of the derived buttons developed for the media player program. These were listed in Section 15.2; they are classes `CXPlayButton`, `CXStopButton`, `CXSearchButton`, and `CXCloseButton`.

### 15.3.1 **CXSearchButton — The Search Button**

The first button to be developed for our media player is the search button. This sits beside text box `m_SearchPath` and should be clicked upon after a user has entered a search path. Once clicked, its `OnClick` event handler should be coded to search the specified path for MP3 files and populate list box `m_FileList`. As it does so, it also adds found files to the playlist of the media player. As shown in the class declaration below, `CXSearchButton` is derived from `CXButton` and overrides an `OnClick` event. It also supports connectivity between controls by maintaining pointers to the text box, list box, and media player. These pointers are set later by the media player application. Following sections examine how this button works.

```
class CXSearchButton : public CXButton
{
private:
    CXTexTbox* m_SearchPath;
    CXListBox* m_FileList;
    CXMediaPlayer* m_MediaPlayer;
    CXPlayButton* m_PlayButton;
protected:
public:
```

```

void OnButtonClick();
CXSearchButton(LPDIRECT3DDEVICE9 Device);
void SetMediaPlayer(CXMediaPlayer* Player) {m_MediaPlayer = Player;}
void SetTextBox(CXTextBox* TextBox) {m_SearchPath = TextBox;}
void SetListBox(CXListBox* ListBox) {m_FileList = ListBox;}
void SetPlayButton(CXPlayButton* Button) {m_PlayButton = Button;}
};
    
```

## 15.3.2 CXSearchButton — Loading Images

Class CXMyMediaPlayerApp could have loaded the images for CXSearchButton itself. But for clarity, the image loading process has been coded into CXSearchButton’s constructor. This means that as soon as this button is instantiated, its images are loaded. The constructor follows.

```

CXSearchButton::CXSearchButton(LPDIRECT3DDEVICE9 Device) : CXButton(Device)
{
    m_SearchPath = NULL;
    m_FileList = NULL;
    m_MediaPlayer = NULL;
    m_PlayButton = NULL;
    SetUnpressedImage("Search.bmp");
}
    
```

## 15.3.3 CXSearchButton — Handling Mouse Clicks

As mentioned, when the search button is clicked it achieves three processes: It searches a specified path for MP3 files, it adds those files to the list box, and it adds those files to the playlist. The following list explains the button click event in more detail. Its function definition follows.

- The search path is taken from text input into the text box. This text box is pointed to by member m\_SearchPath. It could be a string like “C:\mp3s.”
- The search path is then appended with “\\*.mp3” to specify an entire string representing search criterion. This may now look like “C:\mp3s\\*.mp3.”

- MP3 files are then searched for using the `FindFirstFile` and `FindNextFile` Win API functions. See the Win SDK documentation for more information. It works much like getting the first and next items in a linked list, except this represents getting the first and next files at a specified path.
- On each iteration of the loop, a new MP3 file is retrieved. Here, you should add the file to the list box as an item and add the file to the playlist as a song.

```
void CXSearchButton::OnButtonClick()
{
    WIN32_FIND_DATA FileData;
    BOOL bFinished = FALSE;
    std::string Search = m_SearchPath->GetText();
    Search += "\\*.mp3";

    HANDLE hSearch = FindFirstFile(Search.c_str(), &FileData);
    //Get first file

    if (hSearch == INVALID_HANDLE_VALUE)
        return;

    m_FileList->ClearItems();           //Clear list box
    m_MediaPlayer->Stop();             //Stop media player
    m_MediaPlayer->ClearPlayList();    //Clear playlist

    while(!bFinished)
    {
        m_FileList->AddNewItem(FileData.cFileName); //Add list item
        Search = m_SearchPath->GetText();
        Search += FileData.cFileName;

        m_MediaPlayer->AddMediaToPlayList(Search.c_str(), 0); //Add new song

        if (!FindNextFile(hSearch, &FileData)) //Get next file
        {
            if (GetLastError() == ERROR_NO_MORE_FILES)
            {
                FindClose(hSearch);
                bFinished = true;
            }
        }
    }
}
```

```
    }  
    else  
    {  
        FindClose(hSearch);  
        return;  
    }  
}  
}
```

### 15.3.4 **CXPlayButton — The Play Button**

One of the most important buttons for the user is the play button. This is encapsulated into class `CXPlayButton`. It exists in primarily two modes: play mode and paused mode, and a user switches modes by clicking on the button. Clicking for the first time sets the button to play mode, which begins playback of the playlist. Clicking again brings the button into pause mode and consequently pauses playback. Clicking again brings it into play and so on. The class declaration for `CXPlayButton` appears below.

```
class CXPlayButton : public CXButton  
{  
private:  
    CXMediaPlayer* m_MediaPlayer;  
protected:  
public:  
    void OnButtonClick();  
    CXPlayButton(LPDIRECT3DDEVICE9 Device);  
    void SetMediaPlayer(CXMediaPlayer* Player) {m_MediaPlayer = Player;}  
};
```



#### **NOTE**

The code for the constructor is not listed here, but, like all buttons in this chapter, its images are loaded inside this function.

## 15.3.5 CXPlayButton — Handling Button Clicks

As the play button is clicked, it changes modes from play to pause or pause to play. Consequently, the media player acts accordingly. The code below demonstrates how the button click event has been written to handle this appropriately.

```
void CXPlayButton::OnButtonClick()
{
    if (!m_Pressed && m_MediaPlayer)
    {
        m_MediaPlayer->Play(m_MediaPlayer->GetFirstItem());
        m_Pressed = true;
    }
    else
    {
        m_MediaPlayer->Pause();
        m_Pressed = false;
    }
}
```



### NOTE

This event sets the `m_Pressed` member of `CXButton`. You'll remember from Chapter 10 that this ultimately determines which image is drawn, the pressed state (play) or the unpressed state (pause).

## 15.3.6 CXStopButton — The Stop Button

No prizes for guessing what this does. Users who click on the stop button stop media playback. Because this is so simple, the class declaration, constructor, and `OnClick` handler are given below, all in one.

```
class CXStopButton : public CXButton
{
private:
    CXMediaPlayer* m_MediaPlayer;
    CXPlayButton* m_PlayButton;
protected:
```

```

public:
    void OnButtonClick();
    CXStopButton(LPDIRECT3DDEVICE9 Device);
    void SetMediaPlayer(CXMediaPlayer* Player) {m_MediaPlayer = Player;}
    void SetPlayButton(CXPlayButton* Button) {m_PlayButton = Button;}
};

CXStopButton::CXStopButton(LPDIRECT3DDEVICE9 Device) : CXButton(Device)
{
    m_MediaPlayer = NULL;
    m_PlayButton = NULL;
    SetDefaultImage("Stop_Button.bmp");
}

void CXStopButton::OnButtonClick()
{
    if(m_MediaPlayer)
        m_MediaPlayer->Stop();

    if (m_PlayButton && m_PlayButton->GetPressed())
        m_PlayButton->OnButtonClick();
}
    
```

## 15.3.7 CXCloseButton — The Close Button

The final button to be developed for the media player application is the close button. This is by far the simplest of its buttons. All it does is close the application by posting a quit message. This control is so simple, its entire definition can be written as below.

```

class CXCloseButton : public CXButton
{
private:
protected:
public:
    void OnButtonClick();
    CXCloseButton(LPDIRECT3DDEVICE9 Device);
};

CXCloseButton::CXCloseButton(LPDIRECT3DDEVICE9 Device) :
    
```

```

                                CXMediaButton(Device)
{
    SetDefaultImage("Window_Close_Button.bmp");
}

void CXCloseButton::OnButtonClick()
{
    PostQuitMessage(0);
}

```

## 15.4 CXMyMediaPlayerApp — Loading Buttons

---

Having developed the buttons for our media player application, we can now jump back to class `CXMyMediaPlayerApp` and examine how they are loaded. This is achieved with the `LoadButtons` method. Its definition is given below. Notice how the pointers for various controls, such as list boxes, etc., are set using the buttons' accessors.

```

void CXMyMediaPlayerApp::LoadButtons()
{
    m_PlayButton = new CXPlayButton(GetDevice());
    m_StopButton = new CXStopButton(GetDevice());
    m_SearchButton = new CXSearchButton(GetDevice());
    m_CloseButton = new CXCloseButton(GetDevice());

    m_PlayButton->SetXYPos(100,350);
    m_PlayButton->SetMediaPlayer(m_MediaPlayer);

    m_StopButton->SetXYPos(100 + m_PlayButton->GetWidth(), 350);
    m_StopButton->SetMediaPlayer(m_MediaPlayer);
    m_StopButton->SetPlayButton(m_PlayButton);

    m_SearchButton->SetXYPos(m_SearchPath->GetXPos() +
                            m_SearchPath->GetWidth(), m_SearchPath->GetYPos());
    m_SearchButton->SetListBox(m_FileList);
    m_SearchButton->SetTextBox(m_SearchPath);
}

```

```
m_SearchButton->SetMediaPlayer(m_MediaPlayer);
m_SearchButton->SetPlayButton(m_PlayButton);

m_CloseButton->SetXYPos((m_FileList->GetXPos() +
                        m_FileList->GetWidth()) - 32, 15);

m_Window->AddChildControl((CXControl*) m_PlayButton);
m_Window->AddChildControl((CXControl*) m_StopButton);
m_Window->AddChildControl((CXControl*) m_SearchButton);
m_Window->AddChildControl((CXControl*) m_CloseButton);
}
```

### 15.4.1 CXMyMediaPlayerApp — Cleaning Up

Finally, hard work over, the media player application should be released once you've finished using it. This calls the destructor, which looks like the following.

```
CXMyMediaPlayerApp::~CXMyMediaPlayerApp()
{
    if(m_MediaPlayer)
        delete m_MediaPlayer;
}
```

## 15.5 Sample Program — Plugging in the Media Player

Having completed CXMyMediaPlayerApp, it can now be plugged into a standard application. In other words, applications can become media players by just declaring an instance of CXMyMediaPlayerApp, as you would any other control. Furthermore, because it represents a derived CXWindow, it loads all the controls it needs — such as lists, buttons, and more — and implicitly adds them as children. Typically, to become a media player, an application performs the following steps. A sample application can be found on the companion CD.

1. Initialize `CXMyMediaPlayerApp`. This creates the media player application. The code to do this would look like:

```
CXMyMediaPlayerApp g_Application = new CXMyMediaPlayerApp(hWnd,
    g_pd3dDevice);
```

2. Call the `Initialize` method of `CXMyMediaPlayerApp` to load the entire interface, passing it the dimensions of the screen. The controls it loads include buttons, list boxes, and text boxes.

```
g_Application ->Initialize(1024, 768);
```

3. During the rendering loop, pass on the render message. This ensures all controls in the media application are drawn. Remember, `PostMessage` is a method of `CXControl`, from which `CXMyMediaPlayerApp` is ultimately derived.

```
g_Application ->PostMessage(WM_RENDER,0,0,NULL);
```

4. During `WndProc`, ensure all other messages are posted to the control hierarchy, including the customized media player event message.

```
case WM_MOUSEMOVE:
case WM_LBUTTONDOWNBLCLK:
case WM_LBUTTONDOWN:
case WM_LBUTTONUP:
case WM_MBUTTONDOWNBLCLK:
case WM_MBUTTONDOWN:
case WM_MBUTTONUP:
case WM_RBUTTONDOWNBLCLK:
case WM_RBUTTONDOWN:
case WM_RBUTTONUP:
    g_Application->PostMessage(msg,wParam,lParam,NULL);
    return true;
case WM_GRAPHNOTIFY:
    g_Application->Update();
    return true;
```

5. Finally, clean up the object after you're finished.

```
if(g_Application)  
    delete g_Application;
```

## 15.6 Conclusion

---

This chapter really works hand in hand with the companion CD and presents a program that utilizes the work from all three parts of this book. The media player application is not without room for improvement, however. Currently, it does not support media seeking, CD burning, and DVD playback, among other functionality. These are items — with the possible exception of CD burning — that you could add with a bit of reading of the DirectX SDK docs. One particular challenge for advanced readers though, would be to create an entirely skinnable interface. Regardless of which enhancements you may or may not decide to implement, however, the media player thus far represents some definitely sizeable and arguably complex code. One of the best things you can do now is go through various sections of the companion CD to explore and experiment. Hopefully, what seemed confusing as you first began this book will now take shape and seem rather less ominous. As you may notice from this chapter in particular, none of the code is as bad as it first seems.

This page intentionally left blank.

# Afterword

Having read this far, I hope you've gained a few tips and tricks that will help you on your way to creating great interfaces in DirectX. Much of the work presented in this book can, like most things in programming, be improved upon, optimized, and adapted to suit your needs. In presenting the various exercises throughout this book, I have attempted to avoid being too specific or spending too long optimizing code because I wanted to demonstrate the ideas and principles behind interface development. Once you know these, you'll find that you can apply them to almost any interface in DirectX. So, where to go from here? One of the best places to start is the recommended reading section of the book, the appendix. It is also a good idea to check out the source code on this book's companion CD, if you haven't already. There, you'll find a veritable wealth of code just waiting to be tweaked and expanded upon.

I hope this book has been informative and that it can be used as both a reference and a guide. By now, you're probably anxious to put this knowledge into practice — so don't let me stop you. On that note, I'll wish you luck. Happy coding.

This page intentionally left blank.

---

# Recommended Reading

For those interested in learning more about interface development and DirectX in general, I recommend reading the following titles in the order listed.

*3D Math Primer for Graphics and Game Development*

Author: Fletcher Dunn and Ian Parberry

Publisher: Wordware Publishing

ISBN: 1-55622-911-9

*The Zen of Direct3D Game Programming*

Author: Peter Walsh

Publisher: Prima Tech

ISBN: 0-7615-3429-6

*Special Effects Game Programming with DirectX*

Author: Mason McCuskey

Publisher: Prima Tech

ISBN: 1931841063

*Programming Role Playing Games with DirectX*

Author: Jim Adams

Publisher: Prima Tech

ISBN: 1931841098

Appendix: **Recommended Reading**

*Programming Microsoft DirectShow*

Author: Michael Linetsky

Publisher: Wordware Publishing

ISBN: 1-55622-855-4

*Advanced 3D Game Programming with DirectX 9.0*

Author: Peter Walsh

Publisher: Wordware Publishing

ISBN: 1-55622-968-2

# Index

2D coordinates, *see* D3DXVECTOR2  
3D Studio Max, 31

## A

absolute positioning, 116, 126, 133, 146-147, 152, 180, 227  
Acquire function, 83, 90, 96  
ActiveMovie, *see* DirectShow  
Adobe Photoshop, 35, 73-74  
alpha blending (transparency), 45, 51, 61, 72, 76  
alpha channel (transparency mask), 60, 72-75  
alpha tested textures (for skinnable applications), 140  
application window, 140, 148, 161, 181-182  
    dragging, 187  
    minimizing and restoring, 190  
    painting, 186

## B

back buffer, 58, 63, 100, 103, 109, 111, 121  
back end, 2, 321, 323  
base class, *see* base control  
base control, 135, 137, 152  
Begin function, 68  
BeginScene function, 58, 63, 68, 71, 103, 105, 110, 195, 200, 222  
buttons, 7, 204 *see also* CXButton  
    customizing, 322, 327-328, 334-335

## C

canvas, 135, 139-140, 142, 147, 150, 152, 158, 180, 227  
Caps Viewer, 33  
caret, 161, 236-237, 241  
    ID3DXLine as caret, 232  
Cartesian coordinate system, 47-48  
change value (scroll box increment), 252-253  
check boxes, 11, 226, 244 *see also*  
    CXCheckBox

child controls, 140 *see also* hierarchy  
    adding, 143-144  
    clearing, 144  
    counting, 146  
    removing, 145  
child window, 181, 183-184, 322, 324  
    implementing, 184-187  
class hierarchy, 137  
Clear function, 56  
CoCreateInstance function, 292, 296, 297  
CoInitialize function, 292, 294, 296  
COM, *see* CoCreateInstance, CoInitialize, CoUninitialize  
components, *see* controls  
container controls, *see* windows  
controls, 1, 4-5  
cooperative level, 82, 87, 92  
CoUninitialize function, 292  
CreateDevice function, 53  
    creating Direct3D devices, 55, 194  
    creating DirectInput devices, 85-86, 118  
CreateOffscreenPlainSurface function, 59-60, 62, 93  
cstr function, converting strings to char\*, 230  
cursor, 24, 79, 121, 129, 131, 158  
    caret at cursor, 241  
    loading cursor images, 93  
    reading cursor position, 95, 133  
    registering cursors, 94  
    setting cursor position, 94, 96, 240  
    setting cursor visibility, 94  
cursor intersection, 160  
cursor list, 126, 129, 131  
CXButton, 204, 212-219, 244, 251, 328  
    using, 219-223  
CXCheckBox, 244-247  
CXCloseButton, 322, 333-334

## Index

- CXControl, 135, 138, 150-151, 174-176
  - base control and hierarchies, 137, 140-143
  - canvas, 139-140
  - messages, 154-157
  - positioning, 146-148
- CXDropDownList, 250, 283-290
- CXInput, 113, 114-117
- CXKeyboard, 113-114, 117-120
- CXLabel, 204-212, 216
  - using, 219-223
- CXListBox, 250, 268-283, 322
- CXListItem, 263-267
- CXMedia, 307, 308-309
- CXMediaPlayer, 308, 309-317
- CXMouse, 114, 116, 121, 126-134
- CXMouseSurface, 121-122, 125
- CXMyMediaPlayerApp, 320-322
- CXPen, 99, 109
  - deleting, 109-110
  - instantiating, 109-110
  - rendering, 110-111
  - using, 111
- CXPlayButton, 322, 331-332
- CXScrollBar, 250-263 *see also* scroll thumb
- CXSearchButton, 322, 328-331
- CXStopButton, 322, 332-333
- CXSurface, 99-101
  - copying, 102-103
  - deleting, 101
  - instantiating, 101
  - loading images, 102
  - rendering, 104
  - representing back buffer, 103-104
  - using, 104-105
- CXTextBox, 226, 233-243, 322
- CXTexture, 99, 106-107
  - deleting, 107
  - instantiating, 107
  - loading images, 107-108
  - rendering, 108, 110-111
  - using, 111
- CXWindow, 179, 181
  - features, 180
  - using, 193-200
- D**
- D3DCAPS9, 55
- D3DCOLOR, *see* D3DCOLOR\_XRGB
- D3DCOLOR\_XRGB, 57
- D3DPRESENT\_PARAMETERS, 54
- D3DXCreateFont function, 207
- D3DXCreateFontIndirect function, 208
- D3DXCreateLine function, 232
- D3DXCreateSprite function, 66
- D3DXCreateTexureFromFile function, 66
- D3DXLoadSurfaceFromFile function, 61
- D3DXMATRIX, 68
- D3DXMatrixTransformation2D function, 69
- D3DXVec2Add function, 47
- D3DXVec2Scale function, 47
- D3DXVec2Subtract function, 47
- D3DXVECTOR2, 47
- data format, 82 *see also* SetDataFormat
- depth sorting, 170-171
- desktop window, 193
- DIDATAFORMAT, 88
- DIMOUSESTATE, 91, 95, 126
- Direct3D, 29, 45-48
  - applications, 51-52
  - device, 51, 53
  - initializing, 52-53
  - object, 51
  - projects, 48-50
- Direct3DCreate9 function, 52
- DirectAudio, 30
- DirectInput, 29, 80
  - applications, 80-83
  - device, 82, 85, 116-117
  - object, 82, 83, 115
- DirectInput8Create function, 84
- DirectMusic, *see* DirectAudio
- DirectPlay, 30
- DirectShow, 30, 292-293, 307
  - creating applications, 294-295
- DirectSound, *see* DirectAudio
- DirectX, 27-29
  - components, 29-30
  - features, 31-35
  - installation, 37
  - system requirements, 36
- disabling features, 20
- dragging, 187-188
- Draw function, 70-71, 232-233
- drawing
  - lines, *see* ID3DXLine
  - text, *see* ID3DXFont
- DrawText function, 210-211
- drop-down lists, 10, 250 *see also* CXDropDownList

hiding, 287  
 initializing, 286-287  
 showing, 287  
 dynamic string, *see* `std::string`

**E**

`EndScene` function, 58  
 error handling, 19  
 Error Lookup, 32  
 events, triggering, 163, 166, 173

**F**

filter graph, 30, 34, 295  
 focus, 164-165  
 force feedback devices, 80  
`FreeEventParams` function, 305  
 front end, 2

**G**

GDI, 66  
`GetBackBuffer` function, 64  
`GetDeviceCaps` function, 55  
`GetDeviceState` function, 90-91  
`GetEvent` function, 303-304  
 graph builder, *see* `IGraphBuilder`  
`GraphEdit`, 34  
 graphics, 21  
 GUI, 4

**H**

`HandleGraphEvent` function, 305  
 hierarchical posting, 161-162  
 hierarchy,  
   class, 137  
   control, 140  
 Hungarian notation, 42

**I**

icons, *see* graphics  
`ID3DXFont`, 205-212 *see also* `CXLabel`,  
   `CXTextBox`  
`ID3DXLine`, 231-233 *see also* caret  
`ID3DXMesh`, 59  
`ID3DXSprite`, 66, 109 *see also* `CXPen`  
`IDirect3D9`, 51, 53  
`IDirect3DDevice9`, 53  
`IDirect3DSurface9`, 59  
   loading from file, 61  
   *see also* back buffer, `CreateOffscreen-`  
     `PlainSurface`, `CXSurface`

`IDirect3DTexture9`, 65 *see also* `CXTexture`,  
   `ID3DXSprite`  
     loading from file, 66  
`IDirectInput8`, 82, 84 *see also* `CXInput`  
`IDirectInputDevice8`, 85 *see also*  
   `CXKeyboard`, `CXMouse`  
`IGraphBuilder`, 292, 295  
`IMediaControl`, 297  
`IMediaEventEx`, 297, 298  
 immediate data, 90  
 interface, 2  
   design, 16-24  
   examples, 3-4  
 interface flow diagrams, 15  
`IUnknown`, 52, 83

**J**

joysticks, 29

**K**

keyboard, 29  
   configuring, 87-89  
   creating, 86-87  
   reading from, 90-92  
   *see also* `CXKeyboard`, `DirectInput`  
     device  
 keypresses, processing, 238-239

**L**

labels, 8, 204 *see also* `CXLabel`  
 lines, drawing, 232-233  
 linked list, 122-125 *see also* two-way linked  
   lists  
 list boxes, 9, 250, 263 *see also* `CXListBox`  
 list items, 263  
   adding, 274-276  
   clearing, 276  
   *see also* `CXListItem`

**M**

matrix, *see* `D3DXMATRIX`  
 Maya, 31  
 media control, 292, 297  
 media events,  
   catching, 301-302  
   handling, 305  
   reading, 302-303  
 media player, 319-320  
   creating, 323  
   loading buttons, 334-335  
   using, 335-337

## Index

*see also* CXMyMediaPlayerApp  
menus, 12  
Mesh Viewer, 31-32  
message loop, 48, 50, 56, 80, 84  
messages, 154-156  
    handling keyboard, 164  
    handling mouse, 158-162  
    posting, 157  
minimizing, 190-191  
mouse, 29  
    creating, 92  
    processing cursor position, 96-97  
    reading buttons, 97-98  
    reading from, 95-96  
    setting cursor, 93-94  
    *see also* CXMouse, DirectInput device  
MsgProc, 49

**N**  
navigation, 22-23

**O**  
overriding button clicks, *see* customizing  
    buttons

**P**  
page controls, 13  
Paint Shop Pro, 74  
painting, 155, 167  
parent, 140  
parent window, 181-182  
    implementing, 183  
paths, 22-23  
pin, *see* filter graph  
playback, 301, 309  
playlist, 302, 307, 309 *see also* CXMedia  
polling, 83, 132  
Present function, 58

**Q**  
QueryInterface function, 297

**R**  
rectangular drawing space, *see* canvas  
relative positioning, 135, 146-147  
Render function, 50, 52, 56  
RenderFile function, 293, 300  
rendering, 46  
Run function, 301

**S**  
scaling, 106, 108, 205, 257-258  
scaling factor, 260 *see also*  
    D3DXMatrixTransformation2D,  
    D3DXVec2Scale  
scroll bars, 250  
    background, 253  
    *see also* CXScrollBar  
scroll  
    range, 252, 258  
    thumb, 252, 257-258  
    track, 252, 257  
SetCaretPos function, 237  
SetCooperativeLevel function, 87-88  
SetCursorProperties function, 94  
SetDataFormat function, 88  
SetNotifyWindow function, 299  
SetRenderState function, 76  
SetTransform function, 68, 70  
ShowCursor function, 94  
sibling, 140  
skinnable, 140  
snapshot, 56, 83, 90  
std::string, 227-228  
    appending, 229  
    assigning, 228  
    copying, 230  
    editing, 229  
    emptying, 231  
    erasing, 231  
    initializing, 228  
    length, 229  
strcpy function, 209, 228  
strings, converting to char\*, 230  
surface, *see* IDirect3DSurface9

**T**  
tab controls, 13  
target control, 161  
text,  
    drawing, *see* ID3DXFont  
    inserting, 237-238  
    removing, 238  
    setting, 236  
text boxes, 5, 226-227 *see also* CXTextBox  
text edits, 6  
textures, *see* IDirect3DTexture9  
Texture Tool, 35, 75  
tiling, 260-261  
tool tips, 24

two-way linked lists, 141

## U

UI LIB, 136

Unacquire function, 119

UpdateSurface function, 63

user interface, *see* interface

## V

vector, *see* D3DXVECTOR2

vertex, 47

virtual functions, 219

Visual C++, configuring, 39-41

## W

Win 32 SDK, 330

windows, 14 *see also* CXWindow

Windows API, 148, 230

WinMain, 48, 196, 223

WM\_GRAPHNOTIFY, 298

WndProc, 154, 157, 163, 293, 302

wrapper classes, 99

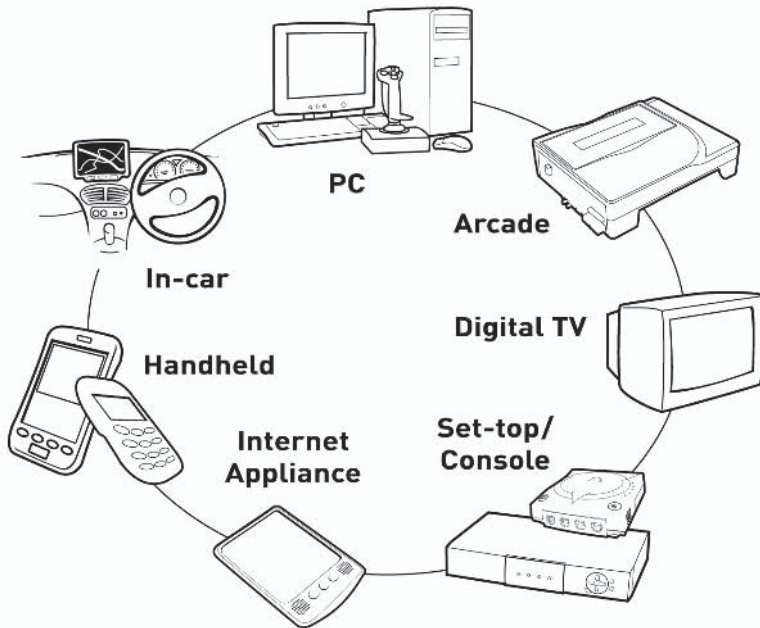
## Z

Z-order, 168-170



# POWERVR

Visionary IP



[www.powervr.com](http://www.powervr.com)

[www.pvrdev.com](http://www.pvrdev.com)

# Get **Deep** into 3D!



## DEEP EXPLORATION.

### Search, View, Translate, Animate, Render and Publish!

**Deep Exploration™** provides easy navigation tools that let you search and view your 2D graphics and 3D models. It gives artists everywhere a production edge, including the ability to quickly translate 2D and 3D file formats with animation included. Deep Exploration creates high quality fast renderings of 3D objects and scenes for use in many graphic applications. It is also a powerful web publication tool, ideal for creating interactive 3D content for web-based presentations.

## DEEP PAINT 3D.

### 3D Painting and Texturing

**Deep Paint 3D®** provides artists with an intuitive, easy to use tool to paint and texture 3D models interactively in 3D! It uses textures or natural media paints which can be brushed directly or projected onto 3D models and scenes. This creative environment supports an integrated workflow with 3ds max®, Maya®, SoftImage® and LightWave 3D®. Deep Paint 3D comes complete with a bi-directional interface to Photoshop® and special support for the Wacom® Intuos™ pressure sensitive tablet.

## DEEP UV.

### Ultimate UV Mapping

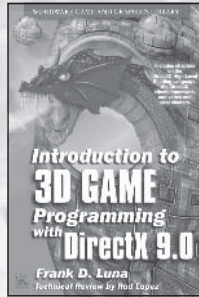
**Deep UV™** is a superior set of tools for the creation and modification of UV mapping for polygonal and sub-division surface models within an interactive 2D and 3D UV mapping environment. Deep UV includes fast and easy to use tools such as soft selection, relax and 3D UV selection. Whether you're a beginner or a professional 3D artist, you'll have the most efficient UV mapping possible with both automatic and advanced features. Deep UV can be used standalone, or with Deep Paint 3D.

**RIGHT** HEMISPHERE.  
[www.righthemisphere.com](http://www.righthemisphere.com)

Check out Wordware's market-  
featuring the following new

## DirectX Game Programming

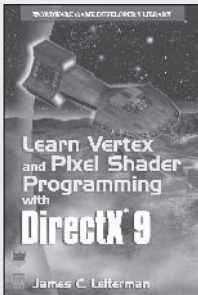
Extensive coverage  
on a wide range of  
DirectX topics.



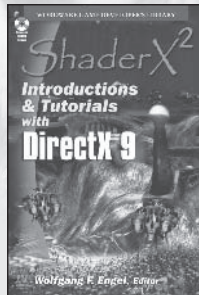
**Introduction to 3D Game Programming with DirectX 9.0**  
1-55622-913-5 • \$49.95  
6 x 9 • 424 pp.



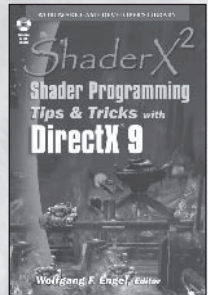
**Advanced 3D Game Programming with DirectX 9.0**  
1-55622-968-2 • \$59.95  
6 x 9 • 552 pp.



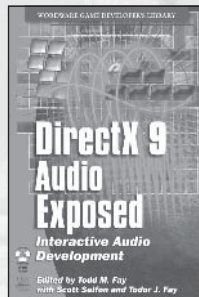
**Learn Vertex and Pixel Shader Programming with DirectX 9**  
1-55622-287-4 • \$34.95  
6 x 9 • 304 pp.



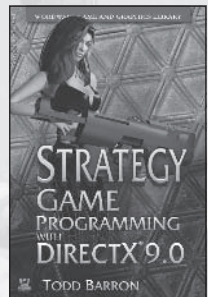
**ShaderX<sup>2</sup>: Introductions & Tutorials with DirectX 9**  
1-55622-902-X • \$44.95  
6 x 9 • 384 pp.



**ShaderX<sup>2</sup>: Shader Programming Tips & Tricks with DirectX 9**  
1-55622-988-7 • \$59.95  
6 x 9 • 728 pp.



**DirectX 9 Audio Exposed: Interactive Audio Development**  
1-55622-288-2 • \$59.95  
6 x 9 • 568 pp.



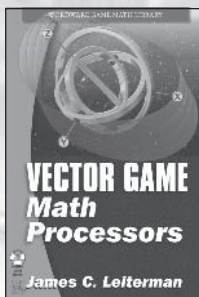
**Strategy Game Programming with DirectX 9.0**  
1-55622-922-4 • \$59.95  
6 x 9 • 560 pp.

# for more?

## Leading Game Developer's Library releases and backlist titles.

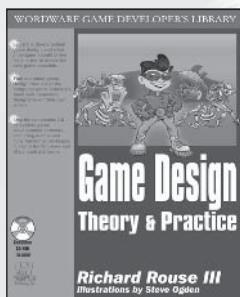


**Java 1.4 Game Programming**  
1-55622-963-1 • \$59.95  
6 x 9 • 672 pp.

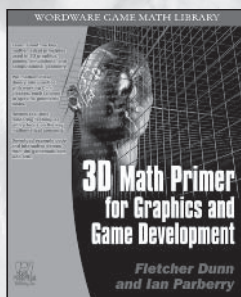


**Vector Game Math Processors**  
1-55622-921-6 • \$59.95  
6 x 9 • 528 pp.

## Game Programming and Production



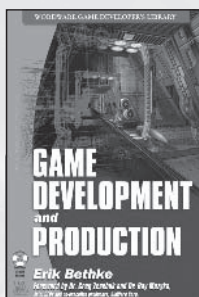
**Game Design: Theory and Practice**  
1-55622-735-3 • \$49.95  
7½ x 9¼ • 608 pp.



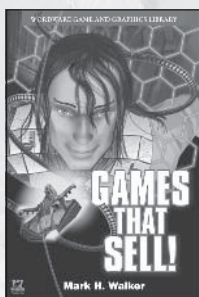
**3D Math Primer for Graphics and Game Development**  
1-55622-911-9 • \$49.95  
7½ x 9¼ • 448 pp.



**Game Design Foundations**  
1-55622-973-9 • \$39.95  
6 x 9 • 400 pp.



**Game Development and Production**  
1-55622-951-8 • \$49.95  
6 x 9 • 432 pp.



**Games That Sell!**  
1-55622-950-X • \$34.95  
6 x 9 • 336 pp.



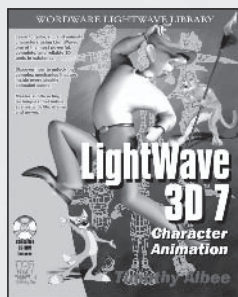
**CGI Filmmaking: The Creation of Ghost Warrior**  
1-55622-227-0 • \$49.95  
9 x 7 • 344 pp.

Use the following coupon code for online specials:

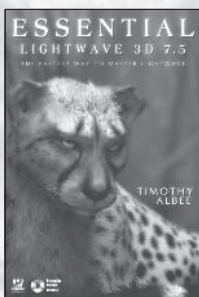
**dxui2491**

# Looking for more?

Check out Wordware's market-leading Game Developer's Library featuring the following new releases, backlist, and upcoming titles.



**LightWave 3D 7 Character Animation**  
1-55622-901-1 • \$49.95  
7½ x 9¼ • 360 pp.

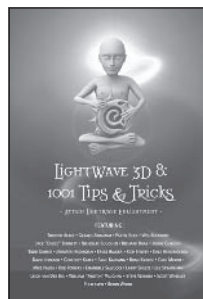


**Essential LightWave 3D 7.5**  
1-55622-226-2 • \$44.95  
6 x 9 • 424 pp.

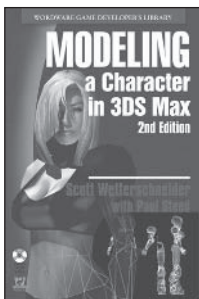


**LightWave 3D 7.5 Lighting**  
1-55622-354-4 • \$69.95  
6 x 9 • 496 pp.

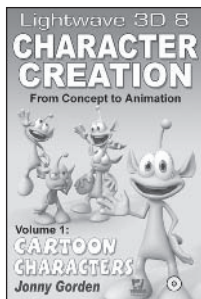
## Coming Soon



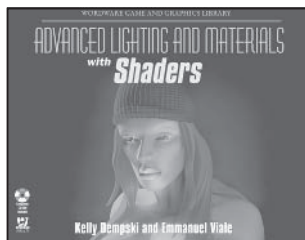
**LightWave 3D 8: 1001 Tips & Tricks**  
1-55622-090-1 • \$39.95  
6 x 9 • 500 pp.



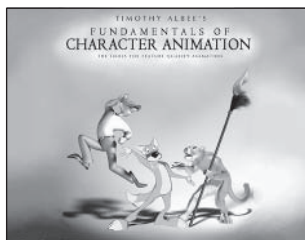
**Modeling a Character in 3DS Max (2nd Edition)**  
1-55622-088-X • \$44.95  
6 x 9 • 550 pp.



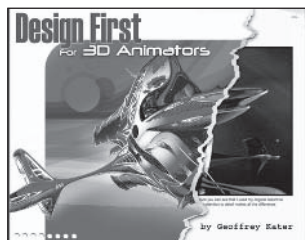
**LightWave 3D 8 Character Creation**  
1-55622-083-9 • \$49.95  
6 x 9 • 500 pp.



**Advanced Lighting and Materials with Shaders**  
1-55622-292-0 • \$59.95  
9 x 7 • 500 pp.



**Timothy Albee's Fundamentals of Character Animation**  
1-55622-248-3 • \$49.95  
9 x 7 • 400 pp.



**Design First for 3D Animators**  
1-55622-085-5 • \$49.95  
9 x 7 • 350 pp.

Visit us online at [www.wordware.com](http://www.wordware.com) for more information.

Use the following coupon code for online specials: **dxui2491**

The Microsoft DirectX 9.0 SDK Update (Summer 2003) was reproduced by Wordware Publishing, Inc., under a special arrangement with Microsoft Corporation. For this reason, Wordware is responsible for the product warranty and for support. If your CD is defective, please return it to Wordware, which will arrange for its replacement. PLEASE DO NOT RETURN IT TO MICROSOFT CORPORATION. Any product support will be provided, if at all, by Wordware. PLEASE DO NOT CONTACT MICROSOFT CORPORATION FOR PRODUCT SUPPORT. End users of this Microsoft program shall not be considered “registered owners” of a Microsoft product and therefore shall not be eligible for upgrades, promotions, or other benefits available to “registered owners” of Microsoft products.

Microsoft DirectX 9.0 SDK Update (Summer 2003) copyright Microsoft Corporation, 2002. All rights reserved.

## CD/Source Code Usage License Agreement

Please read the following CD/Source Code usage license agreement before opening the CD and using the contents therein:

1. By opening the accompanying software package, you are indicating that you have read and agree to be bound by all terms and conditions of this CD/Source Code usage license agreement.
2. The compilation of code and utilities contained on the CD and in the book are copyrighted and protected by both U.S. copyright law and international copyright treaties, and is owned by Wordware Publishing, Inc. Individual source code, example programs, help files, freeware, shareware, utilities, and evaluation packages, including their copyrights, are owned by the respective authors.
3. No part of the enclosed CD or this book, including all source code, help files, shareware, freeware, utilities, example programs, or evaluation programs, may be made available on a public forum (such as a World Wide Web page, FTP site, bulletin board, or Internet news group) without the express written permission of Wordware Publishing, Inc. or the author of the respective source code, help files, shareware, freeware, utilities, example programs, or evaluation programs.
4. You may not decompile, reverse engineer, disassemble, create a derivative work, or otherwise use the enclosed programs, help files, freeware, shareware, utilities, or evaluation programs except as stated in this agreement.
5. The software, contained on the CD and/or as source code in this book, is sold without warranty of any kind. Wordware Publishing, Inc. and the authors specifically disclaim all other warranties, express or implied, including but not limited to implied warranties of merchantability and fitness for a particular purpose with respect to defects in the disk, the program, source code, sample files, help files, freeware, shareware, utilities, and evaluation programs contained therein, and/or the techniques described in the book and implemented in the example programs. In no event shall Wordware Publishing, Inc., its dealers, its distributors, or the authors be liable or held responsible for any loss of profit or any other alleged or actual private or commercial damage, including but not limited to special, incidental, consequential, or other damages.
6. One (1) copy of the CD or any source code therein may be created for backup purposes. The CD and all accompanying source code, sample files, help files, freeware, shareware, utilities, and evaluation programs may be copied to your hard drive. With the exception of freeware and shareware programs, at no time can any part of the contents of this CD reside on more than one computer at one time. The contents of the CD can be copied to another computer, as long as the contents of the CD contained on the original computer are deleted.
7. You may not include any part of the CD contents, including all source code, example programs, shareware, freeware, help files, utilities, or evaluation programs in any compilation of source code, utilities, help files, example programs, freeware, shareware, or evaluation programs on any media, including but not limited to CD, disk, or Internet distribution, without the express written permission of Wordware Publishing, Inc. or the owner of the individual source code, utilities, help files, example programs, freeware, shareware, or evaluation programs.
8. You may use the source code, techniques, and example programs in your own commercial or private applications unless otherwise noted by additional usage agreements as found on the CD.



**Warning:** Opening the CD package makes this book nonreturnable.