

WORDWARE GAME DEVELOPER'S LIBRARY

CROSS-PLATFORM GAME DEVELOPMENT

**MAKING PC GAMES FOR
WINDOWS, LINUX
AND MAC**



ALAN THORN

Cross-Platform Game Development

Alan Thorn

Wordware Publishing, Inc.

Library of Congress Cataloging-in-Publication Data

Thorn, Alan

Cross-platform game development / by Alan Thorn.

p. cm.

Includes index.

ISBN 10: 1-59822-056-X

ISBN 13: 978-1-59822-056-8 (pbk.)

1. Computer games--Programming. 2. Cross-platform software development.

I. Title.

QA76.76.C672T4957 2008

794.8'1526--dc22

2008012132

© 2008, 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: 1-59822-056-X

ISBN 13: 978-1-59822-056-8

10 9 8 7 6 5 4 3 2 1

0805

Linux® is the registered trademark of Linus Torvalds in the U.S. and other countries. Ubuntu is a registered trademark of Canonical, Ltd. Mac is a trademark of Apple Inc., registered in the U.S. and other countries. Windows is a registered trademark of Microsoft Corporation in the United States and other countries.

Other 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

Contents at a Glance

<i>Chapter 1</i>	Cross-Platform Games	1
<i>Chapter 2</i>	Linux Ubuntu and Linux Gaming	49
<i>Chapter 3</i>	Cross-Platform Development Tools	81
<i>Chapter 4</i>	Game Programming Basics	115
<i>Chapter 5</i>	SDL Graphics.	137
<i>Chapter 6</i>	Game Audio	173
<i>Chapter 7</i>	Game Mechanics.	201
<i>Chapter 8</i>	Novashell and 2D Games	225
<i>Chapter 9</i>	Director and Web Games	265
<i>Chapter 10</i>	3D Games with OGRE 3D	299
<i>Chapter 11</i>	Other Cross-Platform SDKs and Tools.	335
<i>Appendix A</i>	GNU Lesser General Public License.	363
<i>Appendix B</i>	BSD License	369
<i>Appendix C</i>	Creative Commons License	371
<i>Appendix D</i>	zlib/libpng License.	379
<i>Appendix E</i>	The MIT License Template	381
<i>Appendix F</i>	STL Public Member Methods Reference	383
<i>Appendix G</i>	SDL Key Codes	389
<i>Appendix H</i>	Novashell Functions	395
<i>Appendix I</i>	Director Events.	407
<i>Appendix J</i>	OGRE OIS Key Codes	409

This book is dedicated to Alfie Stokes,
Emma Gibson, and Lauren Stitt.

Contents

	Acknowledgments	xiii
	Introduction	xv
Chapter 1	Cross-Platform Games	1
	1.1 Platforms	3
	1.1.1 Microsoft Windows	3
	1.1.2 Mac OS X	4
	1.1.3 Linux.	5
	1.1.3.1 Ubuntu.	7
	1.1.3.2 SLAX.	9
	1.1.3.3 Freespire.	10
	1.2 Cross-Platform Games	11
	1.2.1 The Battle for Wesnoth	11
	1.2.2 OpenArena.	12
	1.2.3 UFO: Alien Invasion.	13
	1.2.4 Shockwave and Flash Games.	14
	1.3 Preparing to Go Cross-Platform	15
	1.3.1 Multiple Booting.	17
	1.3.2 Step 1 — Preparing to Multiple Boot	19
	1.3.3 Step 2 — Installing Windows XP.	20
	1.3.4 Step 3 — Installing Windows Vista	27
	1.3.5 Step 4 — Installing Linux Ubuntu	33
	1.3.6 Step 5 — Summary of Multiple Boot	40
	1.4 Virtualization — Simulating Operating Systems	40
	1.4.1 Using VMWare	42
	1.4.2 Creating a Virtual Machine for Linux Ubuntu.	43
	1.5 Conclusion	48
Chapter 2	Linux Ubuntu and Linux Gaming	49
	2.1 Ubuntu Installation and Troubleshooting	50
	2.2 Getting to Know Ubuntu	52
	2.2.1 Ubuntu Login	53
	2.2.2 Ubuntu Desktop	54
	2.2.3 System Monitor	55
	2.2.4 Update Manager.	55
	2.2.5 Screen and Graphics Preferences and Restricted Drivers Manager	56

- 2.2.6 Add/Remove Applications 57
- 2.2.7 Synaptic Package Manager 58
- 2.2.8 Ubuntu Terminal/Console/Shell 59
- 2.2.9 Places | Computer 60
- 2.2.10 Firefox Web Browser 60
- 2.2.11 OpenOffice.org 61
- 2.2.12 Photo Editing. 62
- 2.2.13 Installing and Playing a Game on Ubuntu 63
- 2.3 Linux and “Transgaming” Technologies 65
 - 2.3.1 Cedega 66
 - 2.3.2 CrossOver 67
 - 2.3.3 Wine 68
 - 2.3.3.1 Installing Wine on Linux Ubuntu 68
- 2.4 Automating Ubuntu with Automatrix 70
 - 2.4.1 Installing and Using Automatrix for Linux Ubuntu . . . 71
- 2.5 The Linux Shell 73
 - 2.5.1 Common Shell Commands 74
 - 2.5.2 Creating and Compiling a C Program Using the
Ubuntu Terminal and BASH Shell Commands 79
- 2.6 Conclusion 80

Chapter 3 Cross-Platform Development Tools 81

- 3.1 Code::Blocks. 84
- 3.2 Downloading and Installing Code::Blocks in Ubuntu 86
- 3.3 Downloading and Installing Code::Blocks in Windows 90
- 3.4 Getting Started with Code::Blocks. 95
 - 3.4.1 Code::Blocks Projects. 96
- 3.5 Cross-Platform “Hello World” Application 98
- 3.6 Graphics and GIMP 101
 - 3.6.1 Installing GIMP on Windows or Mac 102
 - 3.6.2 Using GIMP 103
 - 3.6.2.1 Creating Tileable Textures Using GIMP 103
 - 3.6.2.2 Editing Image Transparency Using GIMP 106
- 3.7 Blender 3D 109
 - 3.7.1 Installing Blender 3D on Linux Ubuntu 109
 - 3.7.2 Installing Blender 3D on Windows/Mac 111
- 3.8 Conclusion 113

Chapter 4 Game Programming Basics 115

- 4.1 Game Programming — Getting Started 116
 - 4.1.1 Genre and Objective 117
 - 4.1.2 Time Frame and Budget. 117
 - 4.1.3 Game Ideas. 118
- 4.2 Preparing to Make Games. 122

4.3 Using the STL: Strings and Lists	125
4.3.1 <code>std::string</code>	125
4.3.1.1 Configuring Projects to Use STL and <code>std::string</code> with <code>Code::Blocks</code>	126
4.3.1.2 Declaring, Creating, and Assigning Strings with <code>std::string</code>	126
4.3.1.3 Looping through Characters of a String with <code>std::string</code>	127
4.3.1.4 Searching for Characters in a Specified Instance of <code>std::string</code>	128
4.3.1.5 Extracting and Inserting Substrings from and to a Specified Instance of <code>std::string</code>	129
4.3.1.6 Converting Instances of <code>std::string</code> to Standard <code>char*</code> Pointers.	129
4.3.2 <code>std::vector</code>	130
4.3.2.1 Creating a List with <code>std::vector</code>	131
4.3.2.2 Declaring Instances of <code>std::vector</code>	132
4.3.2.3 Adding Items to a List Using <code>std::vector</code>	132
4.3.2.4 Cycling through Items in a List Using <code>std::vector</code>	132
4.3.2.5 Removing Items from a List Using <code>std::vector</code>	133
4.4 The Game Loop	134
4.5 Conclusion	136
Chapter 5 SDL Graphics	137
5.1 SDL Breakdown	138
5.2 Downloading and Configuring SDL.	140
5.2.1 SDL on Ubuntu.	140
5.2.1.1 Downloading and Installing SDL on Ubuntu Using Synaptic Package Manager	140
5.2.1.2 Downloading SDL Documentation from the Web	142
5.2.1.3 Creating an SDL Project Using <code>Code::Blocks</code> in Linux Ubuntu	143
5.2.2 SDL on Windows.	146
5.2.2.1 Downloading and Installing SDL on Windows	146
5.2.2.2 Creating an SDL Project Using <code>Code::Blocks</code> in Windows	148
5.3 Getting Started with SDL.	152
5.3.1 Initializing and Closing SDL.	153
5.3.2 Creating a Window and Game Loop	154
5.3.3 SDL Surfaces.	159
5.3.3.1 Blitting Surfaces	160
5.3.3.2 Optimizing SDL Surfaces	161

- 5.3.4 Additional File Formats (JPEG, PNG, TGA, and Others) 162
 - 5.3.4.1 Downloading and Configuring SDL Image Development Libraries for Code::Blocks on Ubuntu 163
 - 5.3.4.2 Downloading and Configuring SDL Image Development Libraries for Code::Blocks on Windows 165
 - 5.3.4.3 SDL: Further Image Formats 166
- 5.4 Color Keying with Surfaces 167
- 5.5 Conclusion 169

Chapter 6

- Game Audio 173**
 - 6.1 Recording and Editing Game SFX 175
 - 6.2 SFX Software. 176
 - 6.2.1 Downloading and Installing Audacity on Linux Ubuntu. 178
 - 6.2.2 Downloading and Installing Audacity on Windows or Mac 180
 - 6.3 Recording/Creating and Editing Music 182
 - 6.4 Music Creation Software 183
 - 6.4.1 Downloading and Installing Schism Tracker on Linux Ubuntu 183
 - 6.4.2 Downloading and Installing Schism Tracker on Windows and Mac 185
 - 6.5 Programming Audio with SDL_mixer 186
 - 6.5.1 Installing and Configuring SDL_mixer on Linux Ubuntu. 187
 - 6.5.2 Installing and Configuring SDL_mixer on Windows 189
 - 6.5.3 Initializing the SDL_mixer Library 190
 - 6.6 Sounds and Music with SDL_mixer 192
 - 6.6.1 Loading Music 193
 - 6.6.2 Playing Music 194
 - 6.6.3 Controlling Music 195
 - 6.6.4 Playing Samples through Channels in SDL_mixer . . . 197
 - 6.6.5 Loading Sounds into SDL_mixer as Samples 198
 - 6.6.6 Handling Channels with SDL_mixer 198
 - 6.7 Conclusion 200

Chapter 7

- Game Mechanics. 201**
 - 7.1 Getting Started with Game Worlds 202
 - 7.2 Creating Derivative Objects. 204
 - 7.3 Maintaining Game Objects 205

7.4	Tile-based Levels	207
7.5	Animations and States	211
7.6	Movement	212
7.6.1	Movement with Vectors	213
7.7	Hierarchical Transformations	218
7.8	Z-Order and Depth Sorting	221
7.9	Conclusion	223
Chapter 8	Novashell and 2D Games	225
8.1	Novashell Overview	226
8.2	Downloading Novashell (Windows, Linux, and Mac)	228
8.3	Exploring Novashell Games	230
8.4	Getting to Know Novashell	233
8.4.1	The Game Selection Menu	233
8.4.2	The Editor and Player Modes	234
8.4.3	Getting Started – Loading, Playing, and Editing a Game	235
8.5	Novashell Editor	236
8.5.1	Tile Resources	237
8.5.2	Entity Resources	238
8.6	Novashell Tools	239
8.7	Editing Novashell Levels	241
8.7.1	Selecting, Copying, Pasting, Moving, and Filling Tiles	241
8.7.2	Exploring Maps and Editing Tiles	242
8.8	Creating New Games and Maps	243
8.9	Importing Art into Novashell	245
8.9.1	Importing Files	246
8.9.2	Setting a Player Entity	247
8.9.3	Creating Smaller Tiles from Larger Tiles	247
8.9.4	Setting Collision Information	248
8.10	Novashell System Palette	249
8.10.1	Audio Tiles	250
8.10.2	Color Tiles	251
8.10.3	Invisible Wall Tiles	251
8.10.4	Warp, Waypoint, and Path Nodes	252
8.10.5	Script Tiles	252
8.11	Novashell Scripting	252
8.11.1	Novashell Console	253
8.11.2	Attaching a Script to an Entity	254
8.11.3	Visual Profiles	256
8.11.4	Moving a Character Using the Keyboard	258
8.11.5	Clever Navigation with Pathfinding	261
8.12	Conclusion	264

Chapter 9	Director and Web Games	265
	9.1 Director	268
	9.2 Director Games	269
	9.3 Director and Shockwave Compatibility	271
	9.4 Getting Started with Director	272
	9.4.1 Downloading and Installing Director	272
	9.4.2 Creating an Animated “Hello World” Application in Director	274
	9.5 Director in More Detail	280
	9.5.1 Cast Members	281
	9.5.2 The Stage	282
	9.5.3 The Score Window’s Timeline	284
	9.6 Director Scripting with JavaScript	285
	9.6.1 Frame Scripts	286
	9.6.2 Global Event Scripts	287
	9.6.3 Local Event Scripts	288
	9.7 Practical Scripting	289
	9.7.1 Programming: Shapes, Lines, and Primitives	290
	9.7.2 Printing a List of All Sprites On-stage	291
	9.7.3 Animating Sprites Using Cast Members	292
	9.7.4 Querying Mouse Events	294
	9.8 Using the Projector for Web-based and Stand-alone Games	295
	9.8.1 Building Web Games	296
	9.8.2 Building Stand-Alone Games (EXE for Windows, OSX for Mac)	297
	9.9 Conclusion	297
Chapter 10	3D Games with OGRE 3D	299
	10.1 OGRE 3D	302
	10.2 OGRE 3D Games	304
	10.2.1 Ankh	304
	10.2.2 Other Games	305
	10.3 Installing OGRE 3D	306
	10.3.1 Downloading and Installing OGRE 3D on Ubuntu	307
	10.3.2 Downloading and Installing OGRE 3D on Windows	312
	10.4 Getting Started with OGRE 3D	315
	10.5 Receiving Frame Events	318
	10.6 Adding Objects to a Scene	321
	10.7 Adding Lights and Particle Systems	325
	10.8 Reading User Input with OGRE and OIS	328
	10.9 Conclusion	333

Chapter 11	Other Cross-Platform SDKs and Tools.	335
11.1	Graphics SDKs	338
11.1.1	OpenGL	338
11.1.2	PTK	338
11.1.3	ClanLib	339
11.1.3.1	Installing ClanLib	339
11.2	Audio SDKs	342
11.2.1	FMOD	342
11.2.2	BASS	342
11.2.3	irrKlang	343
11.2.4	Audiere	343
11.2.5	OpenAL	343
11.3	Physics SDKs	344
11.3.1	ODE	344
11.3.2	Newton Game Dynamics	344
11.3.3	True Axis Physics	344
11.3.4	OPAL	345
11.3.5	Bullet	345
11.3.6	PhysX	345
11.4	Network SDKs	346
11.4.1	RakNet	346
11.4.2	HawkNL	346
11.4.3	SDL_net	347
11.5	Artificial Intelligence SDKs	347
11.5.1	Boost Graph Library	347
11.5.2	OpenSteer	347
11.5.3	FANN	348
11.5.4	Garfixia AI Repository	348
11.6	Input SDKs	348
11.6.1	LibGII	348
11.6.2	OpenInput	349
11.7	Scripting SDKs	349
11.7.1	Lua	349
11.7.2	Python	349
11.7.3	Ruby	350
11.7.4	Squirrel	350
11.7.5	AngelCode	350
11.7.6	GameMonkey	351
11.8	Game Engines	351
11.8.1	Torque	351
11.8.2	Irrlicht	351
11.8.3	Game Editor	352
11.9	GUI SDKs	352
11.9.1	OpenGUI	352

11.10	Web SDKs	352
11.10.1	YaBB	352
11.10.1.1	Downloading, Installing, and Creating an Online Forum	353
11.11	Distribution SDKs.	358
11.11.1	NSIS	358
11.11.2	Inno Setup	359
11.11.2.1	Downloading, Installing, and Creating an Installer in Inno Setup.	359
11.12	Conclusion	361
<i>Appendix A</i>	GNU Lesser General Public License.	363
<i>Appendix B</i>	BSD License	369
<i>Appendix C</i>	Creative Commons License	371
<i>Appendix D</i>	zlib/libpng License.	379
<i>Appendix E</i>	The MIT License Template	381
<i>Appendix F</i>	STL Public Member Methods Reference.	383
<i>Appendix G</i>	SDL Key Codes	389
<i>Appendix H</i>	Novashell Functions	395
<i>Appendix I</i>	Director Events.	407
<i>Appendix J</i>	OGRE OIS Key Codes	409
	Index	415

Acknowledgments

This is my fourth book with Wordware and each book has been a pleasure to write. I would like to take this opportunity to thank those people who have helped this book to completion and who have ensured the quality of its contents. My thanks go to:

- Tim McEvoy and Beth Kohler of Wordware Publishing for being both professional and pleasant people to work with.
- Marlies Maalderink for creating the great and colorful cover for this book. Those interested in other work by Marlies can visit her web site at <http://marlies.m3is.nl>.
- My family and friends for their support, encouragement, and understanding.
- And finally, I would like to thank the reader for taking the time to read and study this book. I hope it proves useful.

Alan Thorn (<http://www.alanthorn.net>)

This page intentionally left blank.

Introduction

Browsing the web one cold evening while writing this book led me to a technical web site featuring the following quote, attributed to Herbert Mayer:

“No programming language is perfect. There is not even a single best language; there are only languages well suited or perhaps poorly suited for particular purposes.”

This kind of contextual philosophy — that everything is defined by context, that languages are not in themselves either good or bad, or one thing or another — need not apply only to programming languages. Indeed, it may equally apply to many other facets of computing, from graphic design application to whether one selects an open-source web browser. But perhaps this philosophy applies most of all to the computing concept of “platforms,” meaning operating systems such as Windows, Mac, and Linux. Perhaps platforms too are defined by context and not by themselves.

In the current computing climate, one divided by the controversies of open-source software versus proprietary software, patenting, DRM, and copy-protection scandals, it’s easy to take up positions and arguments for one or the other — believing it to be wholly wrong or wholly right — and then lose sight of the context. But if platforms really are defined by context, by how people use them, the kinds of things one can do on the platform, the kinds of circumstances in which the platform has developed; then to lose sight of context is to lose sight altogether.

It’s quite a simple thing to stand around at a game convention and claim that PCs have become outdated for gaming, giving way to consoles like the Wii, Xbox, or PlayStation; but the reality of course appears not nearly as clear-cut as this. Such simplistic positions ignore the thriving casual game market as found at Reflexive Arcade or Big Fish Games; they ignore the indie-game (or shareware) “resurgence” at sites like GameTunnel, featuring games such as Teenage

Lawnmower, Samorost, Darwina, Jets'n'Guns, or Gish; and they further ignore the increasing “cross-platform” nature of games that run on multiple platforms including Windows, Mac, Linux, *and* the consoles.

Likewise, with an increasingly popular Mac, the changing face of Windows, and the growing community of Linux users worldwide with releases like Ubuntu and PCLinuxOS, the whole PC platform scene has also become the site of conflict and division among many game developers and gamers, with sweeping claims on all sides. Which platform is best? Which is the fastest? The most secure? The easiest? The most customizable? Again, it's quite simple to play the numbers game, mocking one or the other platform by claiming “This platform statistically has a user base of X number of people compared to only Y number for this other platform.” Notice that this is a statement about the platform itself, and not the context. For again, the reality is far less clear-cut than the numbers suggest, if only because not everybody uses one OS; some use many operating systems, on different machines, or dual-booting on the same machine, for different purposes in the same household or in the same office and at different times.

In short, this book attempts to sidestep questions such as “Which is the best?”, “Which platform should I choose?”, and “Isn't this out of date?” Instead, the book accepts that different people use different platforms, and each platform has advantages and disadvantages, many of them determined by context. The cross-platform game developer, then, is not merely someone who scouts around looking for the “best” or most fashionable platform at any given time and simply settles there to make a game for this platform alone; instead, the aim should be to make a game that runs on all platforms. However, the title of this book is *Cross-Platform Game Development*, and this means “cross-platform” in a double sense.

Here, “cross-platform game development” means to develop cross-platform games using cross-platform tools, most of which are also free of charge. Thus, this book examines not just how to make cross-platform games, but how to make cross-platform games using programming editors, graphics suites, and 3D software that are themselves cross-platform.

Who should read this book?

Books are written with a target audience in mind, and here are some typical readers for whom this book is intended:

- Joe studies computers at school, but programs in C++ in his spare time at home and is hoping to make computer games. He's familiar with Windows but no other operating system, and knows his iPod from his MySpace profile, and can distinguish YouTube from Bluetooth, though he doesn't necessarily use them all.
- Anita is a web designer looking to change to a career as an independent game developer selling her games as online downloads from her web site. She's spent almost two years programming in JavaScript and PHP, and can also use Flash. She often uses a Mac and is familiar with Windows, but knows nothing of Linux.
- Kurt is already a professional game developer working for an established software house in a prestigious area of town, but they only make games for consoles. He's thinking of leaving to start a game development business of his own, working from home making cross-platform games. He is familiar with Windows and the consoles, and knows C++, Java, JavaScript, and Lua.

Who should not read this book?

There are some to whom this book may not be suited, though I have no wish to deter anyone who is determined and willing to read along and see what happens. These classic character profiles are guidelines only.

- Alexis is a game enthusiast. She loves playing games on her PC and consoles, though she has never before tried her hand at programming and she really hates math.
- Douglas thinks computer games are okay but perhaps a little nerdy. He did some programming at school and knows the basics but doesn't really enjoy it. He sees game development as an easy route to get rich quick.

What's on the companion files?

The companion files can be downloaded from www.wordware.com/files/gamedev056X. The files include:

- Code samples and projects from the chapters
- Extra game algorithms and routines
- Small game projects

Which game development technologies are considered in detail throughout this book?

- Windows
- Mac
- Linux
- Code::Blocks
- GIMP
- Blender 3D
- STL
- SDL
- SDL_mixer
- Lua
- Adobe Director
- Novashell
- OGRE 3D
- YaBB (briefly)
- Inno Setup (briefly)
- ClanLib (Linux setup and installation)

Chapter 1

Cross-Platform Games

Lewis Carroll’s Humpty Dumpty of *Through the Looking Glass* once said to Alice: “When I use a word, it means just what I choose it to mean, neither more nor less.” For poor Alice, or anyone who invests hope in having a conversation where the listener can be entirely certain of what the speaker means, Humpty’s response must have been somewhat disheartening. In a world where words mean anything the speaker chooses, anything goes. The word “modern” is one such example. One can speak about *modern history*, beginning around the sixteenth century; *modern art* and *modern furniture*, around the turn of the twentieth century; *modern languages* to mean French, German, and English; and *modern day* to mean the present moment. Since the advent of computers, it can be said with certainty that there are more words floating about than there were before, and the meanings of words change from context to context.

Computers are themselves modern inventions, and have engendered a whole new set of terms and languages such as C++ and Leet, which is the Internet language that includes terms like LOL and ROFL. There are also new terms, from bits and bytes to RAM and ROM. Even more recently, the language of computing is colored by underlying themes of friendliness and togetherness. Software is “interactive” and “accessible,” hardware is “compliant” and “compatible,” and the relationship between users and computers is “responsive” and “productive.”

“Cross-platform” is one of the most common and fashionable computing terms among software developers, though it is by no means a new term. There are cross-platform games, office applications, 3D rendering software, and thousands of other products stacked as high as the eye can see on store shelves, or ready to download online at the touch of a button. And yet — despite its prevalence throughout

computing generally — the term cross-platform suffers the fate of Humpty Dumpty’s words in that it is ambiguous, meaning different things to different developers and different consumers, and at different times. For those spending their hard-earned cash on the latest software to call itself cross-platform, consumers could be forgiven for assuming they can know what to expect.

Cross-platform, then, is a term that requires one to read between the lines to get to the bottom of its meaning. The most basic definition of cross-platform software is: software that can run on multiple platforms. The word “platform” is itself another term usually interchangeable with “operating system”; so examples of a platform include Windows, Mac, Linux, and others. Thus, cross-platform software seems to be software that runs on Windows, Mac, Linux, and others. But this is not always the case. Some software is more or less cross-platform than others; some support up to ten different operating systems and others only two, and the operating systems that are supported by a specific product often reflect a political choice on behalf of the developers. Some older products have claimed to be cross-platform while supporting only Windows 98 and Windows 2000 (claiming each to be a separate platform); and more recently cross-platform has come to be thought of as software that runs on both Mac and Windows. It should, however, be pointed out that cross-platform doesn’t usually mean the *same* application (the same executable) runs on each platform. Rather, the same application (source code) is compiled to different executables (called *distributions*), one executable for each platform. Furthermore, developers tend to sell each distribution separately, meaning their product must be purchased twice if the user wants both a Windows and a Linux version. So in summary, cross-platform software as commonly recognized today, and cross-platform games specifically, are distinguished as follows:

- Games that run on at least two *different species* of operating system. That might be Windows and Mac; Mac and Linux; or Windows, Linux, and Mac. But different versions of the same species are not considered multiple platforms, such as: Windows 98, Windows Me, Windows XP, and Windows Vista. These are together one platform. Of course, most cross-platform software is compliant with only selected versions from a species of operating system.
-

- Games that are compiled into different distributions (one executable per platform) that are often sold separately. This, however, is not a requirement of cross-platform software so much as the “norm” and the inevitable by-product of the differences between platforms. There are, as we shall see, exceptions to this rule, and these exceptions are becoming more common.

1.1 Platforms

Cross-platform games are those that run on at least two (ideally more) different species of platform: Windows, Mac, and Linux. Let’s consider each more closely.



NOTE. There are, of course, other platforms besides Windows, Mac, and Linux, but this book focuses on only these three.

1.1.1 Microsoft Windows

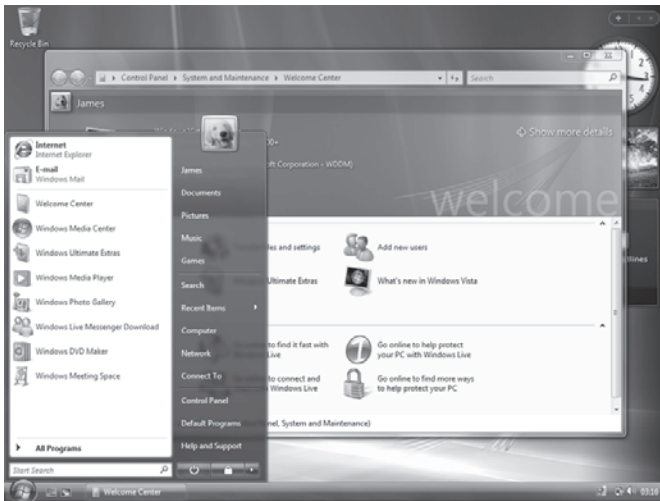


Figure 1-1: The Windows Vista desktop

Windows needs no introduction and needs no justification of its importance to commercial gaming. It is singularly the most famous and

prevalent among the three platforms considered in this book, and as a species is composed of a long line of versions together estimated in 2004 to have been used by around 90% of the client market worldwide, and in 2007 by 92.86%. Though these statistics can be deceptive and likely overestimate the dominance of Windows (as we shall see), it is nevertheless the single most dominant of all the platforms available to the masses of consumers today. The first version of Windows (Windows 1) arrived in 1985, and its descendants can be traced through Windows 95, 98, Me, XP (still the most widely used version at the time of writing), and the more recent (and controversial) Windows Vista. It perhaps goes without saying that since Windows is the most widely used platform by a considerable margin, most computer games that are developed ensure as a priority that Windows is among their supported platforms.



NOTE. This book assumes the reader has a familiarity with Microsoft Windows. And accordingly, the basics of how to install and use Windows are not discussed in this book.

1.1.2 Mac OS X



Figure 1-2: The Mac OS X desktop

The term *Mac* now refers to a range of computers such as MacBooks and iMacs, and many of them feature the Mac OS X (pronounced Mac OS 10) operating system. Each version of Mac OS X is named after a

big cat, including Jaguar, Panther, Tiger, and now Leopard (released in 2008). The Mac OS generally comes preinstalled with Mac computers, and its growing popularity makes Mac an important platform for games. Later chapters of this book consider Mac games in more detail.

1.1.3 Linux

Linux has in recent years grown and attracted intense interest from a diverse population of users and developers at an almost unprecedented pace, with an estimated user base of over 25 million. Linux is a Unix-inspired operating system that is open-source and free of price, and is thus considered to be free software. However, “Linux” is now an umbrella term used to refer to a whole range of different operating systems that are adaptations based on the Linux source code, and these adaptations are individually called *Linux distributions* (or distros). First, however, some terms need to be clarified, and these are considered in the following sidebar.

Linux is a foundational operating system that is open-source and free software, and is used as the starting point for many other derivative (distros) operating systems that are also often open-source and free software. This means many Linux-based distros are complete operating systems that are free of price to download and feature media player facilities, Internet browsing software, office suite applications, and more. So Linux is an exciting platform for developers and users since most distros are community run, free, and continually changing, and furthermore, most Linux distros can be installed *alongside* Microsoft Windows. Installing and configuring multiple operating systems is considered in more detail later in this chapter.

Next, let’s discuss a few of the Linux distributions, particularly those that have attracted a growing community of game players and developers.

Open-Source and Free Software

Linux — along with a selection of other software — is said to be *open-source*, which refers to the openness of software design, or the ability for developers and users alike to access the source code of their software, learn from it, and change it. Thus, open-source software refers to software that makes public its source code, laying bare its inner workings like an open book, and is open to scrutiny and investigation.

Linux is also said to be *free software*, which is different from software that is free of price. Software that is free of price is free insofar as it costs nothing (in terms of money) to use. Examples of software that is free of price (but which isn't "free software") include: Internet Explorer, avast! Anti-Virus Home Edition, and the Opera web browser. Free software, on the other hand, is often free of price but is more than this. According to the Free Software Foundation (<http://www.fsf.org>), for software to be free software it must offer users the following four "freedoms":

1. Freedom to run the software for any purpose. That is, users are free to choose how they use their software.
2. Freedom to study and adapt the software to suit the user's needs. This effectively means the software must be open-source.
3. Freedom to redistribute copies of the software to other users, either with or without charge.
4. Freedom to "improve" the program and release changes back into the community for others to use. This condition relies on all three prior conditions.

Examples of free software are GIMP (photo editing program), Firefox web browser, and Blender 3D. Other free software is listed at <http://directory.fsf.org/>, which is maintained by the Free Software Foundation and the United Nations Education, Scientific and Cultural Organization (UNESCO).

1.1.3.1 Ubuntu

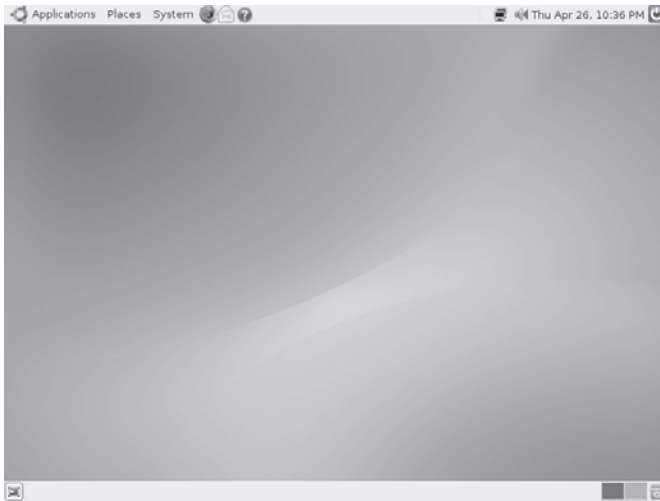


Figure 1-3: The Ubuntu desktop

Ubuntu, generally recognized as the fastest growing desktop-oriented distribution of Linux, takes its name from the sub-Saharan humanist ideology that promotes openness, equality, and relationships between people. Owned by Canonical Ltd. and developed by a community, the Ubuntu distribution is open-source and free software, and features a media player, web browser, office suite, photo editing software, e-mailing facilities, and a comprehensive database of other free software to download. In 2007 Linux DistroWatch (<http://www.distrowatch.com>) ranked Ubuntu as the most “searched for” Linux distribution online, and its popularity among users reflects the effort Ubuntu developers make to ensure Ubuntu is simple to use and compatible with a wide selection of hardware. The web site for Ubuntu is <http://www.ubuntu.com>, where a CD installer can be downloaded for free.

Ubuntu is perhaps unique among other distros for attracting a significant proportion of game players; for this reason the Ubuntu community has seen an exciting and relatively steep increase in the number of games available compared to other Linux distributions. Later sections of this chapter consider some popular cross-platform games currently available. Overall, Ubuntu in the past few years has proven itself a popular distribution and an important contender as a Linux gaming platform. Consequently, it will be discussed in further detail in later sections and chapters of this book.

Downloading and Burning an Ubuntu ISO File

Linux Ubuntu — like many applications — can be downloaded as an ISO (optical disc image file) CD/DVD, which is an archive file containing within itself the entire contents of a disc, similar to the way in which a ZIP file contains other files. The contents of the ISO can be burned directly to a disc using most CD burning software, such as the InfraRecorder application that is specifically for burning ISOs. InfraRecorder can be downloaded free of charge from <http://infra-recorder.sourceforge.net/>.

To burn an Ubuntu ISO:

1. Download the Ubuntu ISO image from the official Ubuntu home page.
2. Download and install InfraRecorder from <http://infrarecorder.sourceforge.net/>.
3. Insert a blank CD/DVD and select **Actions | Burn Image** from the InfraRecorder's main menu. Then select the Ubuntu image file.

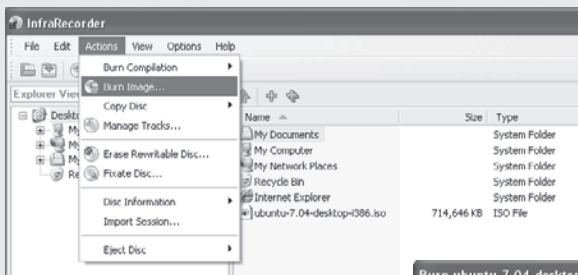


Figure 1-4

4. Select a slow CD writing speed (to reduce risk of erroneous burning) and click **OK**.

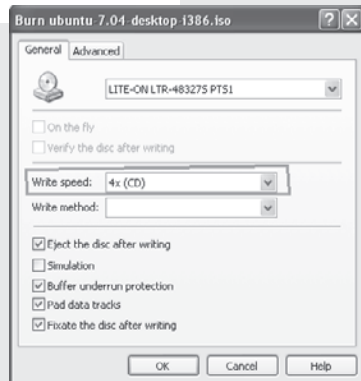


Figure 1-5

1.1.3.2 SLAX

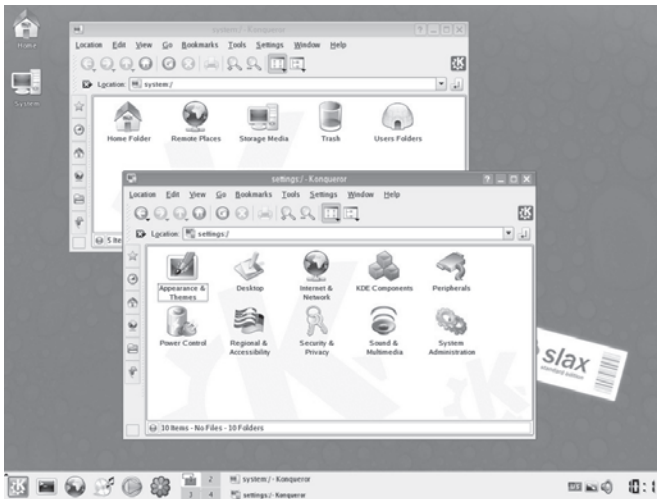


Figure 1-6: SLAX

An OS in miniature, SLAX is important and distinguished from other Linux-based distributions, as well as other operating systems, because it markets itself as a portable, pocket-sized operating system powerful enough to be a desktop environment but small enough to fit onto a USB stick. It is a slick operating system users can carry with them that can be plugged in, booted up, and used on any computer wherever they go without the need for permanent installation, and lasting only as long as the USB remains in the port. Though SLAX is not as prevalent as distros like Ubuntu or Freespire, nor has it been the focus for an exodus of gamers, it is nonetheless a distro of growing popularity not least because of its quirks and simplicity. For this reason SLAX has the interesting potential to be a platform for “gamers on the go” — those who carry their games with them and wish to resume playing whenever the opportunity arises, regardless of the computer that is available. There are several SLAX versions that can be downloaded for free at <http://www.slax.org/>.

1.1.3.3 Freespire

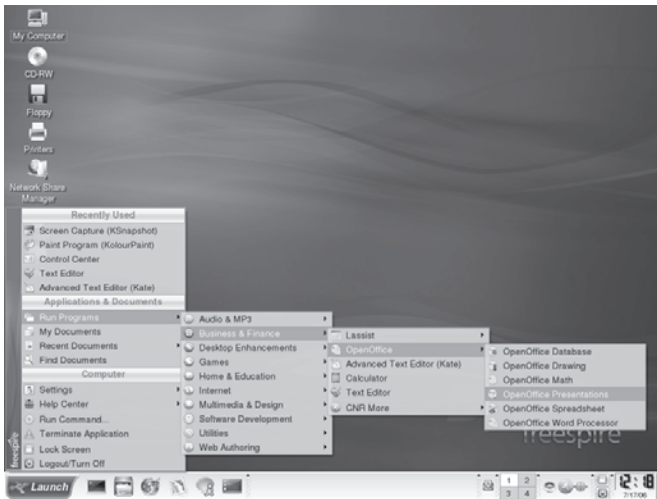


Figure 1-7: Freespire

Based on Ubuntu, Freespire is open-source and composed from free software, though it also offers users (controversially) a selection of non-free software. Freespire is thought by some to be the most “Windows-like” of the Linux distros, and is hailed as being a first step into Linux for those migrating from a Windows platform. But Freespire is a complete and exciting distro offering wireless Internet, office suites, web browsers, and media playing, and boasts a growing community of gamers and game developers. It shall not be considered further in this book, but readers would be well advised to investigate this distro further. It can be downloaded from <http://www.freespire.org>.

1.2 Cross-Platform Games

Cross-platform games may potentially run on Windows, Linux, and Mac. This section explores some cross-platform games that are full versions, open-source, and free to download and play.

1.2.1 The Battle for Wesnoth

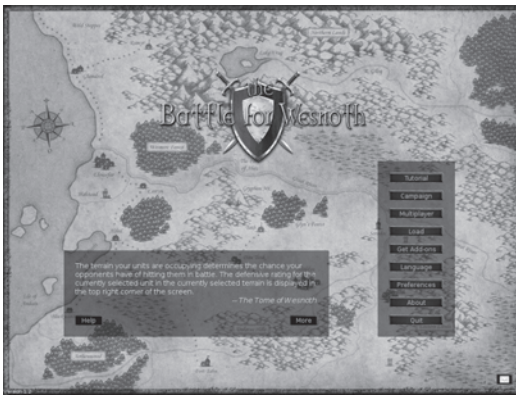


Figure 1-8: The Battle for Wesnoth



Figure 1-9

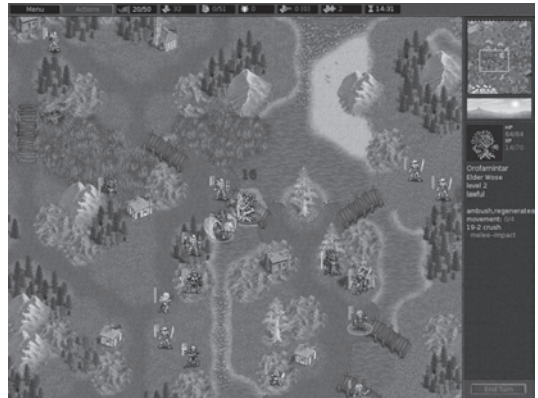


Figure 1-10

Supporting Windows, Linux, and Mac, The Battle for Wesnoth is a cross-platform and free software turn-based strategy game set on a hex-grid. Multiple players, or a single player and computer AI, take turns in tactically deploying mythical creatures and units across

fantasy environments such as forests and deserts, combatting each other to complete campaigns. The game is developed in C++ and uses a series of cross-platform game development kits (including SDL, explained later) to ensure compatibility with multiple operating systems. Beyond supporting Windows, Linux, and Mac, it also claims to support other operating systems, including AmigaOS 4, BeOS, FreeBSD, NetBSD, OpenBSD, Solaris, RISC OS, and GP2X. The Battle for Wesnoth can be downloaded from <http://www.wesnoth.org/>.

1.2.2 OpenArena



Figure 1-11: OpenArena

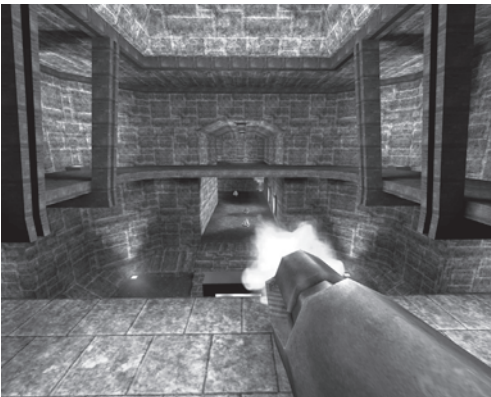


Figure 1-12

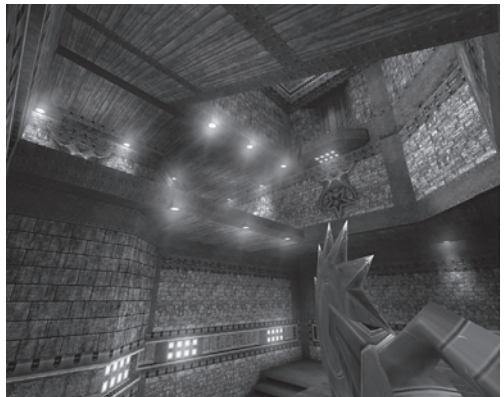


Figure 1-13

Quake Arena and Unreal Tournament fans might be delighted to know there is a free, open-source, and cross-platform FPS tournament game called OpenArena. Based on the Quake III engine, OpenArena features

big guns and big environments, inside which one or more players and AI opponents battle it out to be the last man or woman standing. OpenArena supports a selection of different platforms including Windows, Linux, and Mac, and can be downloaded from <http://www.openarena.ws/?about>.

1.2.3 UFO: Alien Invasion



Figure 1-14: UFO: Alien Invasion



Figure 1-15



Figure 1-16

Inspired by the old series of X-COM strategy games, and later games such as UFO: Aftermath, the free, open-source game UFO: Alien Invasion is about protecting the Earth from aliens. In command of a paramilitary rescue force intended to save the planet, the player must construct a headquarters, deploy a team of agents to intercept UFOs, and help a team of scientists reverse-engineer captured alien

technology. UFO: Alien Invasion is a vast and complex turn-based strategy game supporting Windows, Linux, and Mac OS X. It is developed primarily with OpenGL, an SDK (software development kit) for creating fast-paced, cutting-edge 3D games. OpenGL is explained later in this book. UFO: Alien Invasion can be downloaded from <http://ufoai.ninex.info/>.

1.2.4 Shockwave and Flash Games

The web is awash with Shockwave and Flash, two cross-platform technologies developed by Adobe for embedding multimedia content — from graphics to sound — in web pages and also for distributing as stand-alone executables. Shockwave and Flash offer developers a set of easy-to-use tools for producing *animated* and *interactive* content, and for this reason have been popular choices for the creation of games by an exciting new generation of online game developers. Diner Dash, Home Run Rally, The 13th Doll, and Samorost are but a few among thousands of online games developed using Shockwave or Flash, and a further selection of such games can be found at <http://www.shockwave.com>. This book will later focus on Shockwave, and will explain how Shockwave games are distinct from games developed using other technologies, and how this distinction brings advantages and limitations to both game developers and players. More information regarding Flash and Shockwave can be found at the Adobe site at <http://www.adobe.com>.

1.3 Preparing to Go Cross-Platform

To summarize, the operating system for which a game is designed is called the target platform; games that are “cross-platform” are those that can execute on *two or more different* species of target platforms (such as Windows, Linux, or Mac OS X). Examples of cross-platform games include The Battle for Wesnoth, Diner Dash, and OpenArena, among thousands of others. For developers looking to develop cross-platform games, it therefore follows that each developer must have the facilities (hardware and software) to test their game for bugs on each target platform before releasing it to users of that platform as a final product. In other words, if a developer creates a cross-platform game for Windows and Linux (Ubuntu), then the developer must also have the facilities to test the game on those platforms. Cross-platform game development — the ability to develop and run a single game on multiple platforms — can occur in primarily one of three arrangements, depending on the budget and preferences of the developer. We’ll take a quick look at each, then consider them in further detail.

- **Multiple computers** — Perhaps the most obvious but most expensive setup is to purchase several computers, one for each platform, and install the respective operating systems on each machine. This is generally regarded as a “keep things simple” approach, which provides a no-nonsense 1-to-1 correspondence between operating system and hardware (one machine, one OS) where each OS has complete control of a single machine. Testing a game in this setup means to install, compile, and run the game on each machine.

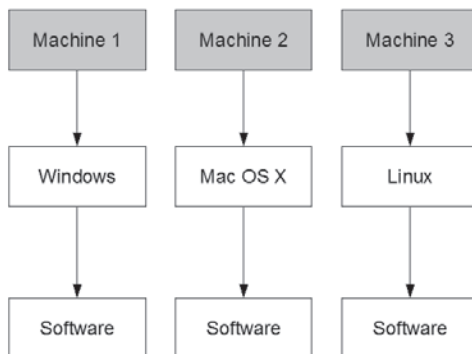


Figure 1-17: Multiple machine diagram

- **Multiple booting** — Widely considered the cheapest but most complicated method is to create a series of hard disk partitions on a single machine and install each operating system on a different partition, selecting from a menu which OS to boot from at system power-on. In short, the computer is switched on, and the user chooses which operating system to use for this session. This means the machine must be restarted each time a change of operating system is required.

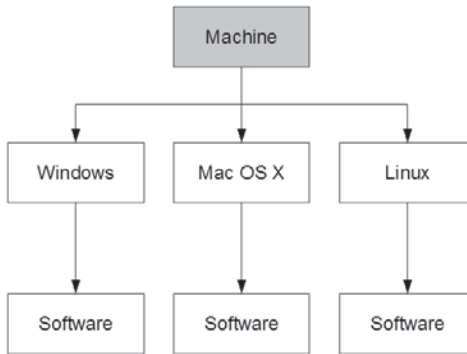


Figure 1-18: Multiple boot diagram

- **Virtualization** — The online encyclopedia Wikipedia defines “virtualization” as the “abstraction of computer resources” and as “a technique for hiding the physical characteristics of computing resources.” For the purpose of cross-platform games, however, *virtualization* can be thought of us as a form of software that allows other machines and platforms to be simulated (or emulated) on a single machine and inside a single operating system. Thus, virtualization is arguably the simplest and most convenient cross-platform solution because multiple machines and operating systems can be run and emulated (including several platforms running simultaneously in different windows) at the click of a button on a single machine without a need to restart between changes of operating system.
-

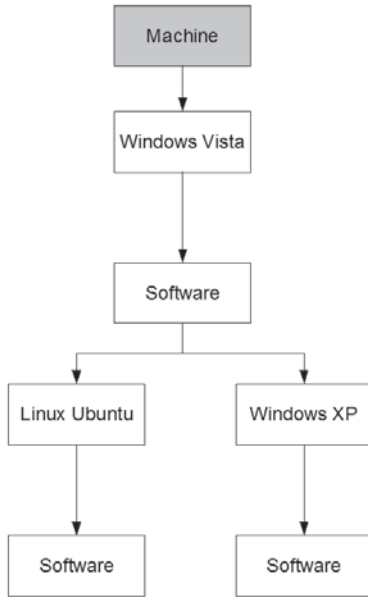


Figure 1-19: Virtualization diagram

1.3.1 Multiple Booting

In a multiple booting setup, two or more operating systems coexist on the same machine. More accurately, each operating system is installed on a different hard disk partition on the same machine, and only one operating system is selected by the user at power-on to use for a single session. This book focuses on dual-booting a machine with Windows and Linux installed; that is, Windows XP or Windows Vista, and Linux (Ubuntu). This step-by-step guide further assumes that the computer to configure for multiple booting begins with a clean hard drive (i.e., no operating system is installed), and that the user has a bootable CD/DVD or bootable USB version of both Windows (XP or Vista) and Ubuntu. For instructions on how to clear a computer (how to format a hard disk), please consult the following sidebar.

Formatting Hard Disks

```

Darik's Boot and Nuke beta.2003052000
----- Options -----
Entropy: Linux Kernel (urandom)
PRNG:   Mersenne Twister (mt19937ar-cok)
Method: 0x0 5228-22_M
Verify:  Last Pass
Rounds:  1
----- Statistics -----
Runtime:  00:00:21
CPU Load: 96%
Throughput: 5973 KB/s
Limiter:  Disk I/O
Errors:   0

(IDE 0,0,0,-,-) Upware Virtual IDE Hard Drive
[04.33%, round 1 of 1, pass 1 of 7] [writing] [5973 KB/s]

```

Figure 1-20: Boot and Nuke

Starting with a blank machine — an empty hard disk — means clearing out existing data, and this occurs through formatting. Before formatting, however, readers are advised that *all data existing on the hard disk before formatting may be irretrievably lost*. For this reason, it is highly recommended that all data is archived and backed up to a safe storage device. For example, data can be burned to a CD/DVD or copied to a USB stick.

The format process removes all data from a hard disk, leaving it blank and fresh to receive new operating systems and information. To perform the format process, the freely available and bootable Darik's Boot and Nuke application can be used. Small enough to fit on an old 1.44 MB floppy disk, Darik's Boot and Nuke is one among many applications designed especially for formatting disks, and can be downloaded from <http://dban.sourceforge.net/> as either an EXE file or an ISO image burnable straight to CD/DVD. (See Section 1.1.3.1 for more details on burning an ISO file.)

Boot the computer with this CD/DVD (in other words, start the computer with this CD inserted into the drive) and follow the on-screen instructions. At the command prompt of Boot and Nuke, users can enter the `autonuke` command to format all writeable disks attached to the computer.

1.3.2 Step 1 — Preparing to Multiple Boot

Given a computer with a formatted hard disk, one that is clean of all information, the following steps illustrate the process of installing multiple operating systems to different partitions on the hard disk. Here we'll discuss a dual-boot configuration featuring either Windows XP and Ubuntu, or Windows Vista and Ubuntu. Users installing Windows XP should proceed to Step 2, and users installing Vista can skip to Step 3. Step 4 considers the installation of Linux Ubuntu, and Step 5 considers the final details of the dual-boot configuration.

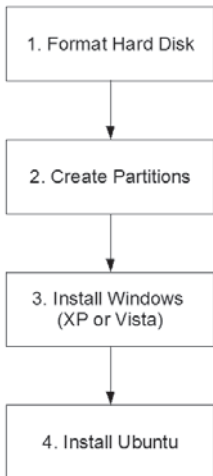


Figure 1-21: Installation diagram

1.3.3 Step 2 — Installing Windows XP

The following steps illustrate the installation process for Windows XP.

1. Insert the Windows XP CD/DVD into the CD drive and reboot the computer.
2. Depending on a computer's configuration, the boot loader may automatically boot from the Windows XP CD, or it may require prompting. If the latter, boot from the CD and the Windows XP installer will start automatically.

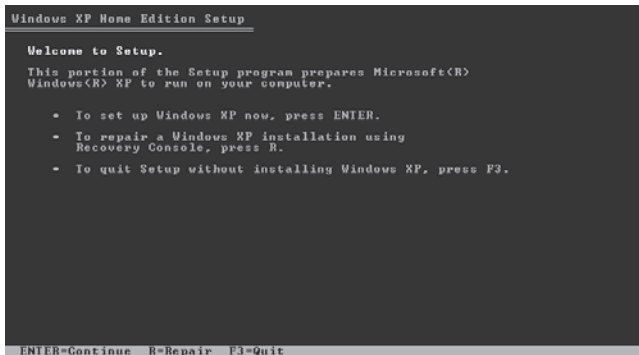


Figure 1-22: Windows XP Installation Setup screen

3. Press **F8** to accept the EULA (end-user license agreement).

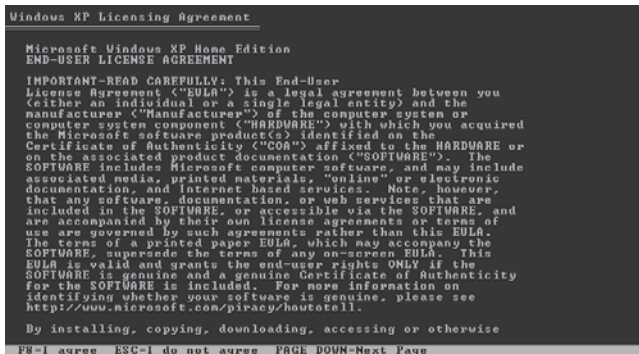


Figure 1-23: Windows XP Licensing Agreement screen

- The Windows XP Partition screen will appear, allowing the user to divide the hard disk into partitions; that is, to divide the disk space into separate drives. The formatted hard disk will appear to the installer as one contiguous sequence of unpartitioned bytes. Press the **C** key to create a partition.

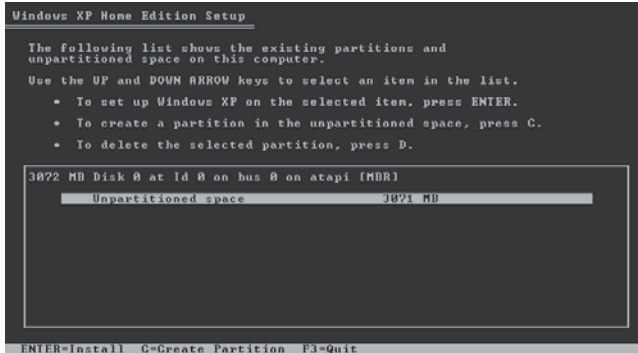


Figure 1-24: Windows XP Partition screen

- Enter the size in megabytes (MB) of the partition to be created by the installer. Remember, one operating system is allocated to each partition. The recommended size of this Windows XP partition, therefore, is half the total size available since another partition must later be created for hosting Linux Ubuntu. Press **Enter** to complete.

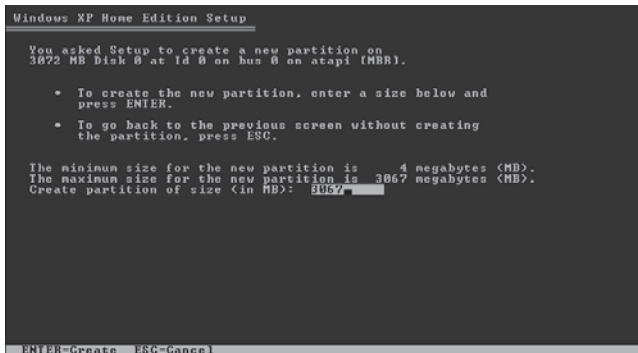


Figure 1-25: Windows XP Partition confirmation screen

- Select the partition, and the installer prepares to format it in one of two file systems: FAT or NTFS. Select **FAT** (not NTFS), and the standard format (not quick). FAT is selected because Ubuntu has

the ability to read from and write to files on FAT partitions, whereas Linux support for NTFS file systems is more limited.

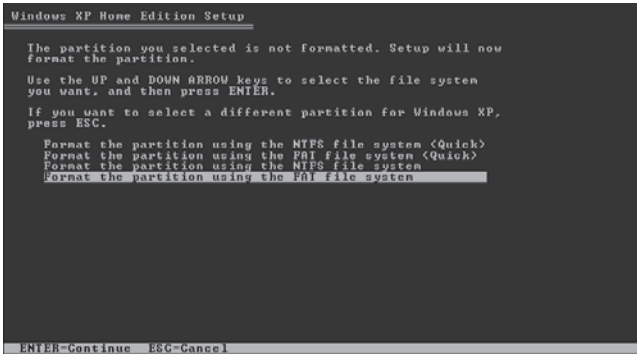


Figure 1-26

7. When the installer confirms your selection, press **Enter**.



Figure 1-27

8. The installation begins, and the process is measured on the progress bar.

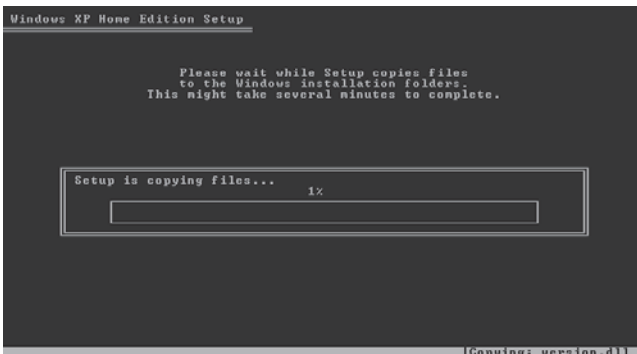


Figure 1-28

9. During and after this process, the computer may restart on one or more occasions, but eventually the GUI installer will appear.

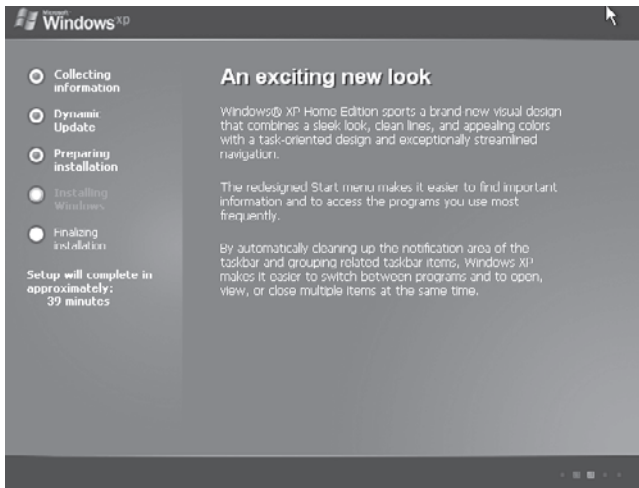


Figure 1-29

10. The localization dialog appears, prompting the user to select a local language. This choice affects the keyboard layout, among other linguistic stylizations for Windows XP. Click the **Customize** button to change the system language. The default language is English (US).

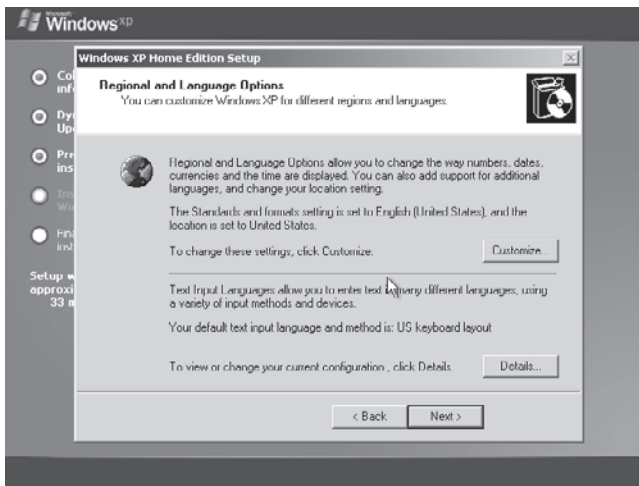


Figure 1-30

11. Enter your name and organization as applicable.

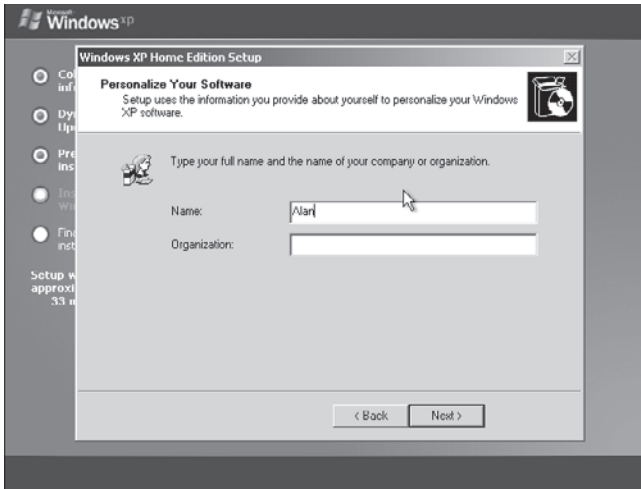


Figure 1-31

12. Enter the Windows XP product key.

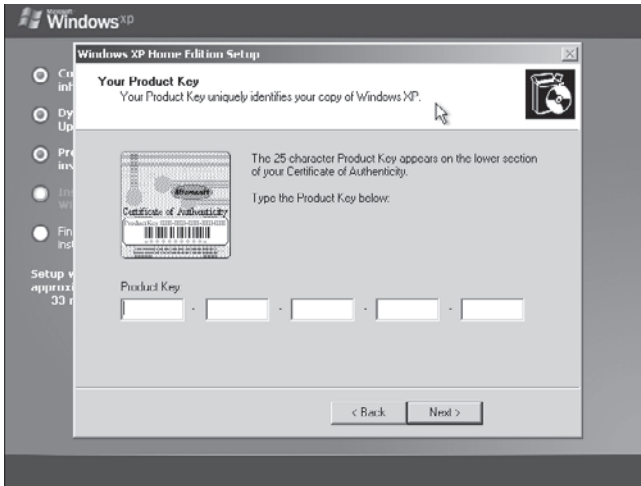


Figure 1-32

13. Select the date and time.

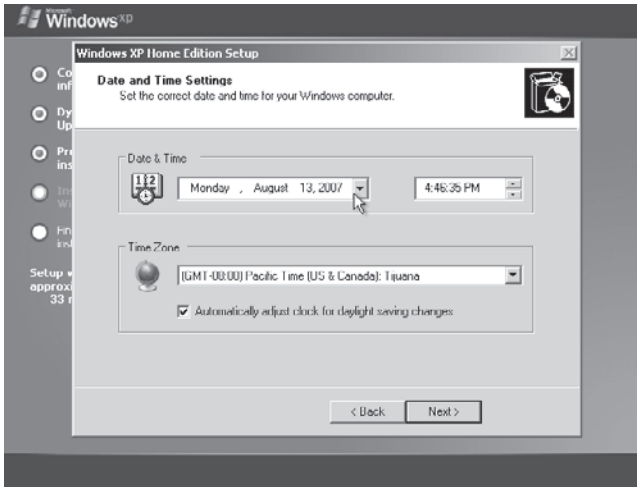


Figure 1-33

14. Select the **Typical settings** option for the network settings unless the computer has specific requirements.

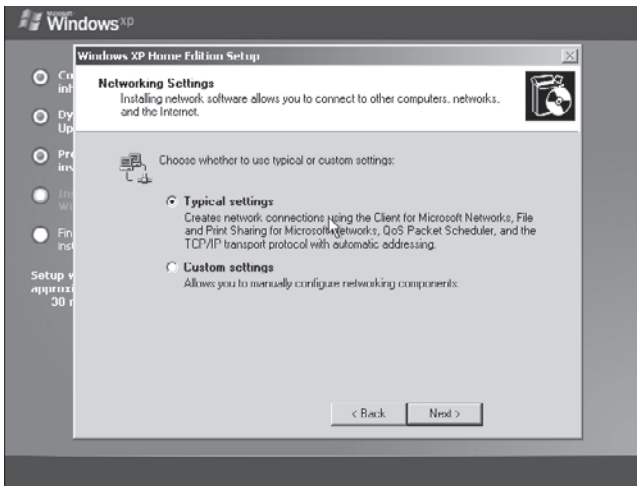


Figure 1-34

15. The computer might restart, and then the configuration wizard appears. Click **Next**.



Figure 1-35

16. Windows XP installation is complete. Go to Step 4.



Figure 1-36

1.3.4 Step 3 — Installing Windows Vista

The following stages illustrate the installation process for Windows Vista.

1. Insert the Windows Vista DVD into the DVD drive and reboot the computer.
2. Depending on the computer's boot loader and configuration, the Windows Vista installer may begin automatically, or it may require prompting. If the latter, boot from the DVD and the installation begins.

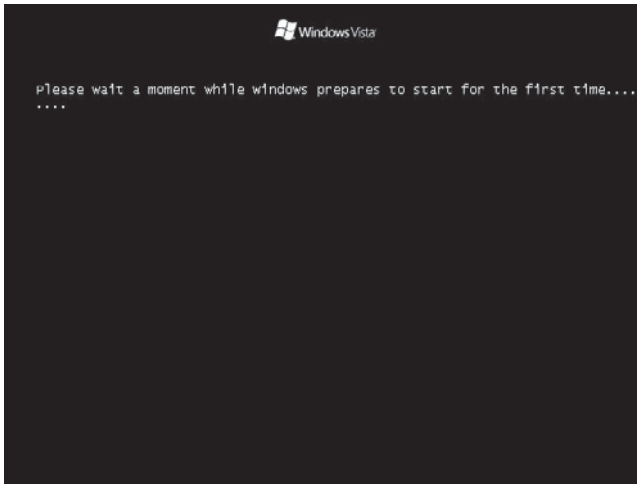
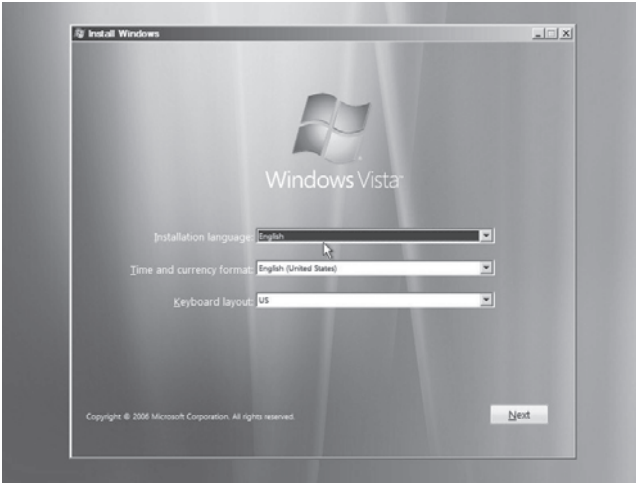


Figure 1-37

3. Set the language settings (default: English (US)), and click **Next**.

*Figure 1-38*

4. Click **Install now**.

*Figure 1-39*

5. Enter the product key and click **Next**.

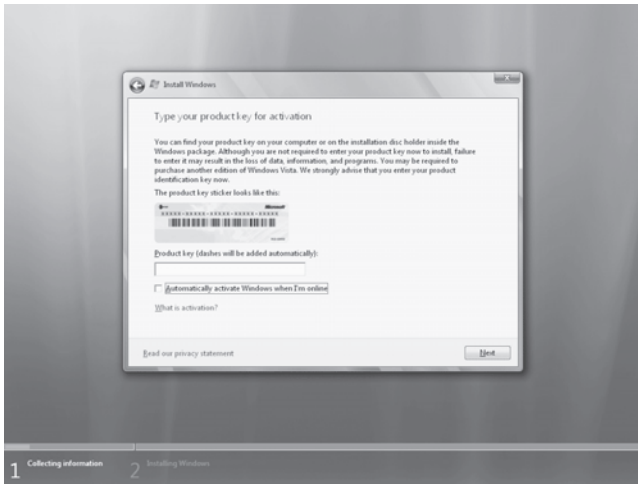


Figure 1-40

6. Click the check box to accept the EULA (end-user license agreement) and click **Next**.
7. The Windows Vista partition screen will appear, allowing the user to divide the hard disk into partitions; that is, to divide the disk space into separate drives. The formatted hard disk will appear to the installer as one contiguous sequence of unpartitioned bytes. Click the **New** button to create a new partition, and enter the size in megabytes (MB) of the partition to be created by the installer. Remember, one operating system is allocated to each partition. The recommended size of this Windows Vista partition, therefore, is half the total size available since another partition must later be created for hosting Linux Ubuntu. Click **OK**. The installer will format the partition if required.

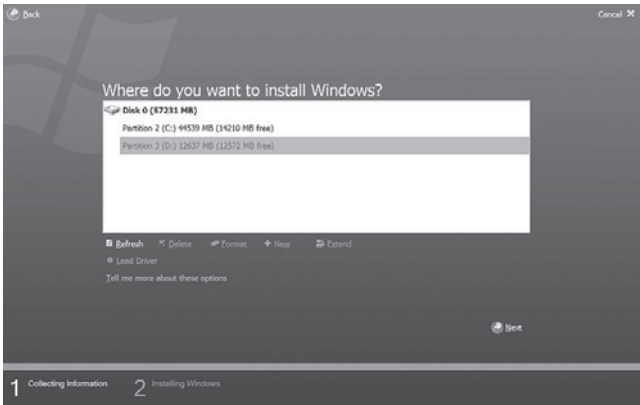


Figure 1-41

8. Click **Next** once you've verified the install settings.

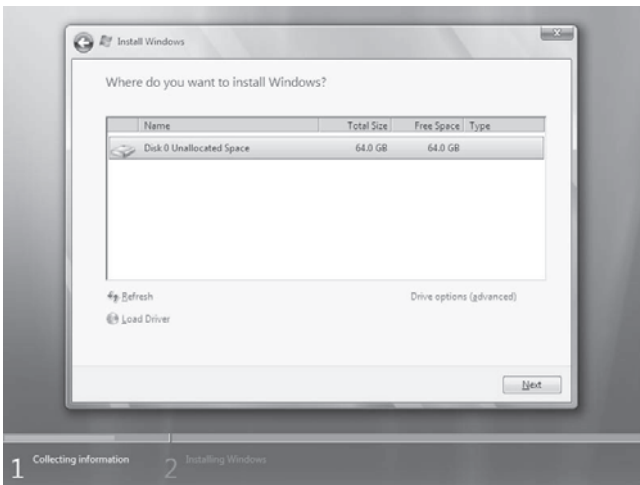


Figure 1-42

- The installation progress window appears, outlining the installation procedure.

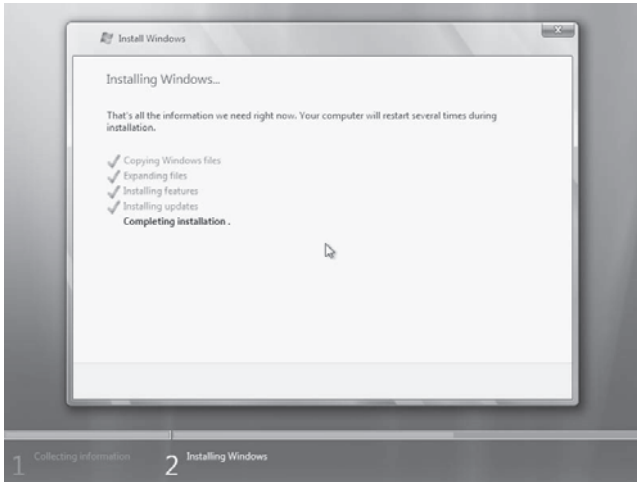


Figure 1-43

- Once completed, the computer might restart on one or more occasions, and finally the user details dialog appears. Enter a user name and password, and select a profile image. Click **Next**.

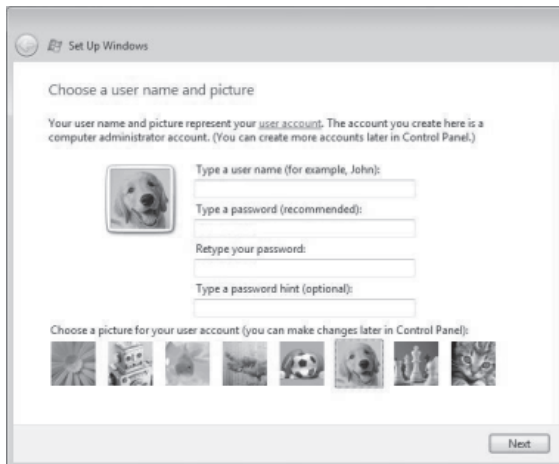


Figure 1-44

11. Enter a computer name and click **Next**.

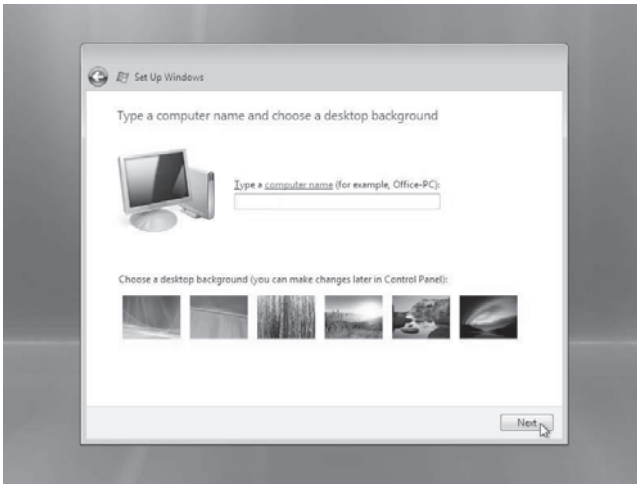


Figure 1-45

12. Select the **Use recommended settings** option for the Internet and protection configurations.

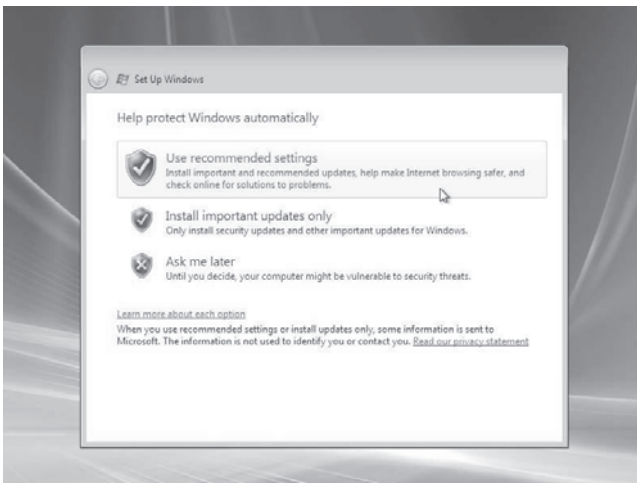


Figure 1-46

13. Enter the time and date details and click **Next**.

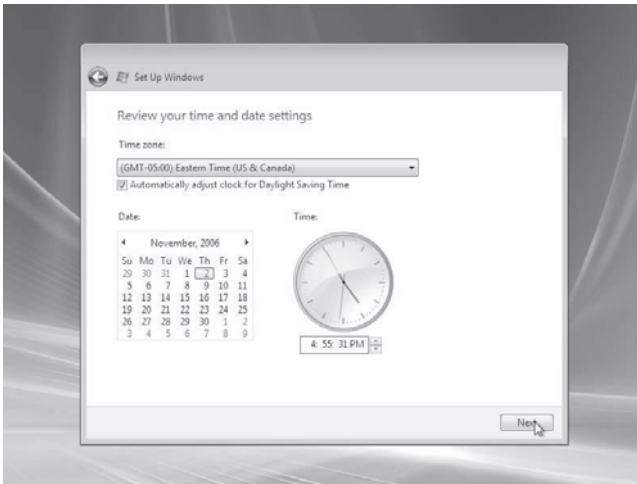


Figure 1-47

14. Click **Start**, and installation is complete. Go to Step 4.

1.3.5 Step 4 — Installing Linux Ubuntu

The following steps illustrate the installation process for Linux Ubuntu.



NOTE. The downloaded Ubuntu ISO is typically burned and installed from a CD/DVD; however, Ubuntu may also be installed directly from the ISO file using WUBI, a Windows-based ISO installer designed specifically for Ubuntu. WUBI can be downloaded for free at <http://wubi-installer.org/>.

1. Insert the Ubuntu CD (created in Section 1.1.3.1) in the CD-ROM drive and reboot the computer.
2. Select the **Start or install Ubuntu** option from the menu, and Ubuntu will start in Live mode.



Figure 1-48

3. Ubuntu Live mode is designed to be a “test drive,” or a “no obligation, try before you install” mode, allowing users to inspect, preview, and use the features of Ubuntu without any obligation to install. Since Live mode installs nothing to the computer, users can safely reboot without any configuration changes having occurred. To permanently install Ubuntu, double-click the **Install** icon on the desktop and select a language. Then click **Forward**.

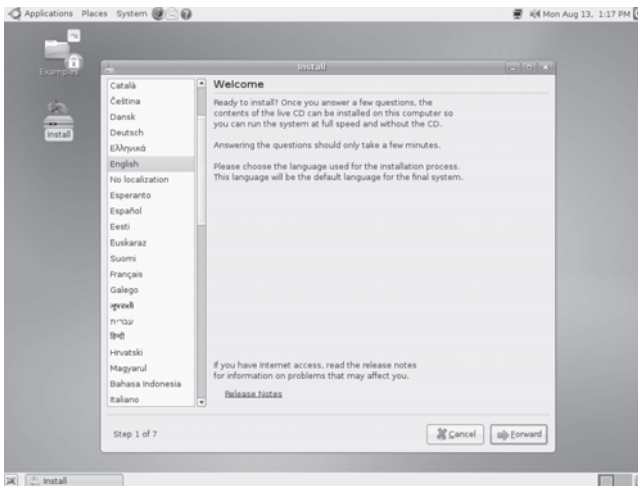


Figure 1-49

4. Select a location from the drop-down list or by clicking on the world map, and then click **Forward**.



Figure 1-50

5. Set the localization for this operating system, which includes features such as keyboard layout and other linguistic stylizations, and then click **Forward**.

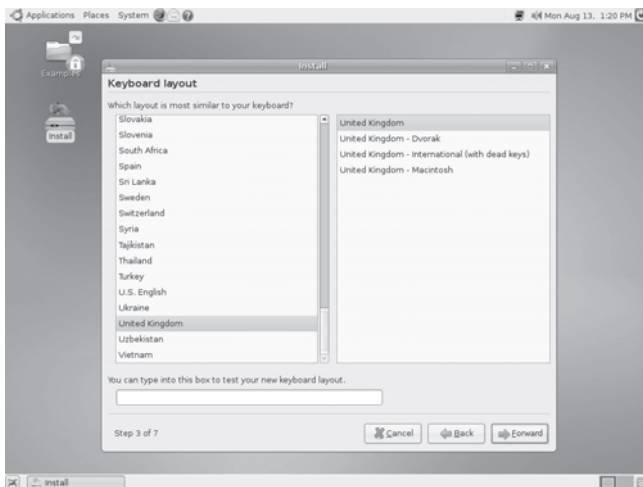


Figure 1-51

6. Select **Manual** for the partition setting. Click **Forward**.



Figure 1-52

7. The partition manager allows users to create, delete, and edit partitions. Click **New partition** to create a new partition onto which Ubuntu will be installed.

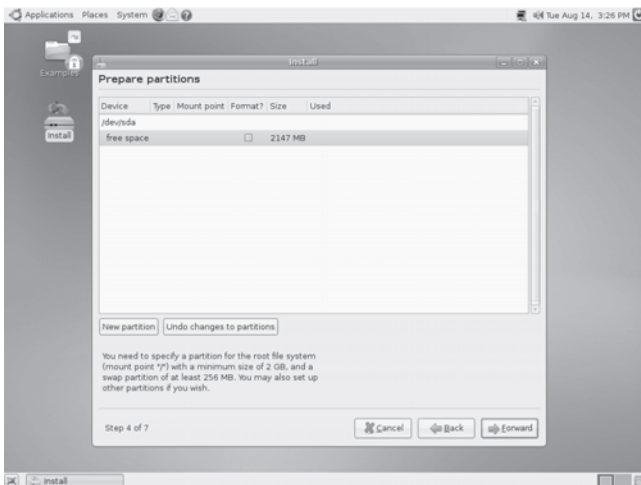


Figure 1-53

8. Specify the size in megabytes and the format of the file system for the partition to create, and associate the partition to a “mount point” (analogous to a drive in Windows). Click **OK**. The primary

mount point in Linux Ubuntu is “/”. The Ubuntu operating system is discussed in further detail in the next chapter.

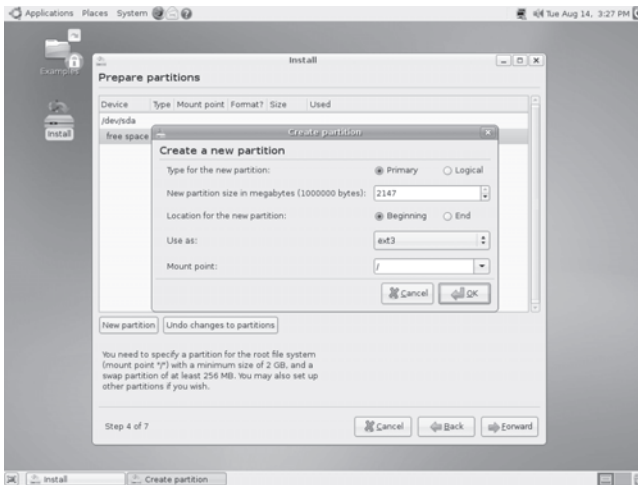


Figure 1-54

9. The Ubuntu partition manager can also be used to create additional partitions such as a Swap partition from any available space to improve the performance of Ubuntu. Add extra partitions if required, and then click **Forward**.

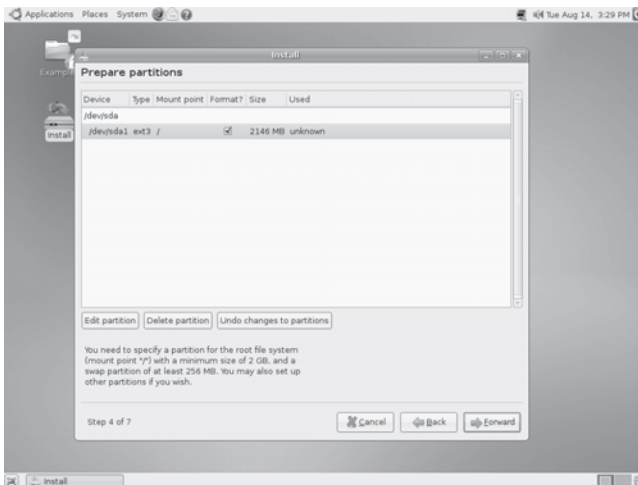


Figure 1-55

10. The migration wizard can be used to import e-mail, messages, address books, and other information and settings from a Windows system. Using the check boxes, select whatever information (if any) needs to be imported, and click **Forward**.

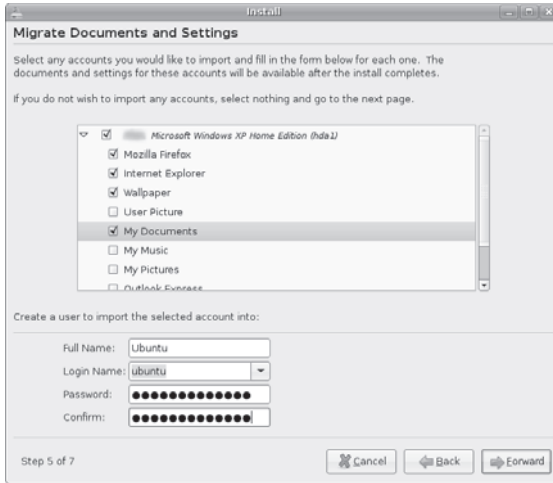


Figure 1-56

11. Enter your user name and password. Click **Forward**.

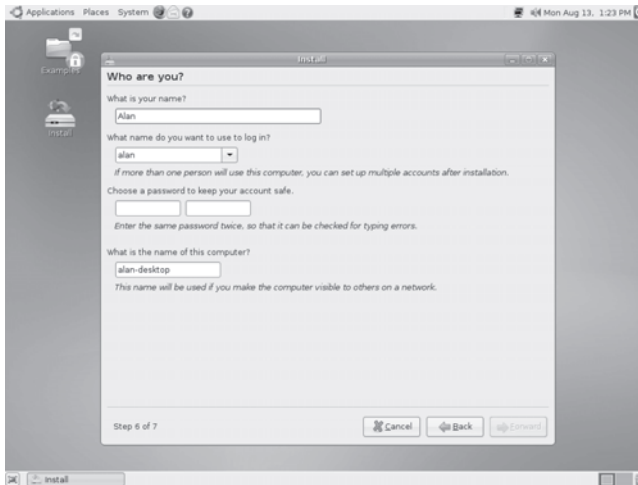


Figure 1-57

12. Confirm the installation details and click **Install**.

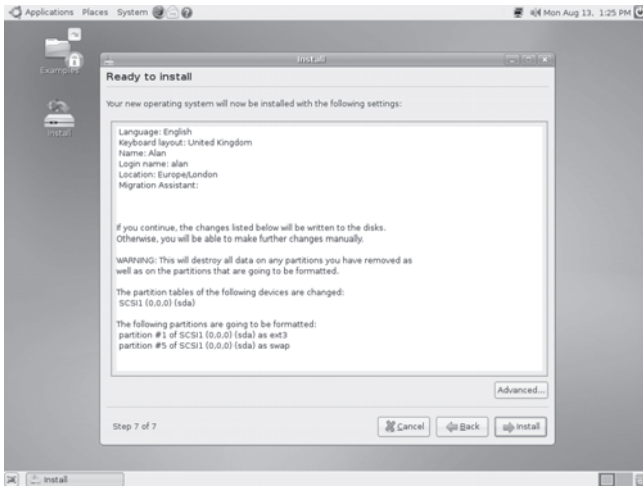


Figure 1-58

13. Ubuntu will install and restart when completed. Go to Step 5.

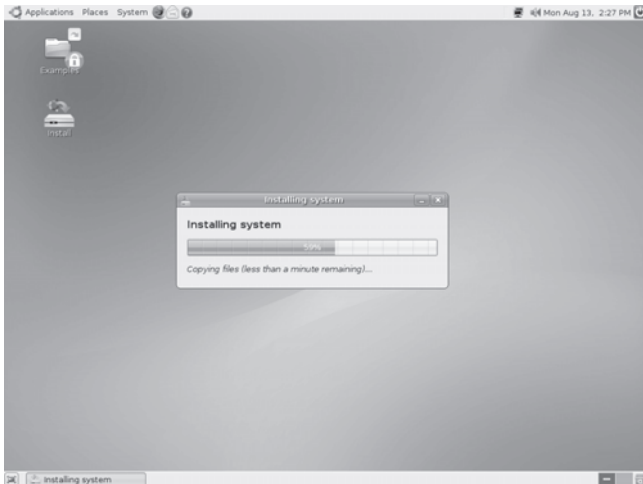


Figure 1-59

1.3.6 Step 5 — Summary of Multiple Boot

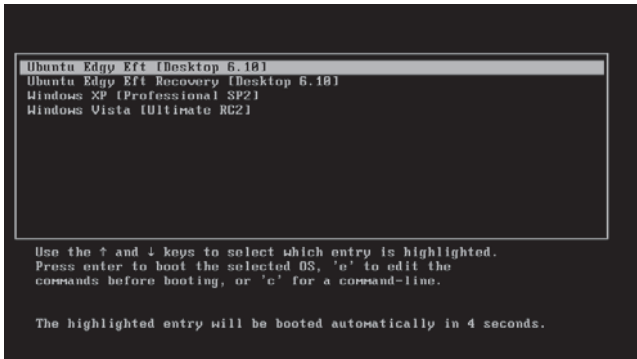


Figure 1-60

Multiple boot systems are those with two or more operating systems installed on different partitions, and the example in this chapter demonstrated how to configure a dual-boot machine featuring either Windows XP and Ubuntu, or Windows Vista and Ubuntu. Restart the computer to test the configuration, and at boot-up the GRUB boot loader menu appears, offering a choice of operating system — Windows or Linux — to boot to. The dual-boot configuration is now completed.

1.4 Virtualization — Simulating Operating Systems

The famous matryoshka dolls (or Russian nesting dolls) contain smaller copies of themselves inside one another, and those smaller dolls inside contain still more smaller copies inside themselves, and so on in a sequence of progressively smaller dolls. In short, what the Russians did for dolls, virtualization does for operating systems. By abstracting a computer’s hardware, virtualization software makes it possible for a *host* operating system to “contain” one or more other client operating systems within itself, simulating each *guest* OS as

though it were installed and executing on a different computer. Among the diverse range of virtualization software available, two of the most well known are Microsoft Virtual PC and VMWare Workstation.



NOTE. Host refers to the primary operating system running on the user's machine. Guests refer to the simulated operating systems running within the host using virtualization software.

- **Virtual PC 2007** — Virtual PC is a Microsoft-developed virtualization product freely available for select versions of Windows, and is designed to simulate other operating systems, primarily other versions of Microsoft Windows, although other operating systems such as DOS and OS/2 are also supported. Virtual PC 2007 can be run on the following host operating systems: Windows Vista Business, Windows Vista Enterprise, Windows Vista Ultimate, Windows Server 2003 Standard Edition, Windows Server 2003 Standard x64 Edition, Windows XP Professional, and Windows XP Tablet PC Edition.
- **VMWare Workstation 6** — VMWare Workstation is a virtualization product that lacks the Microsoft-centricity of Virtual PC, and consequently it supports a greater variety of both host and guest operating systems, including Windows and Linux. Unlike Virtual PC, however, VMWare is a commercial product that can be purchased under a variety of different arrangements. An evaluation version of VMWare, and more information about VMWare generally, can be found at the VMWare web site at <http://www.vmware.com/>. This book focuses on virtualization using VMWare.

1.4.1 Using VMWare

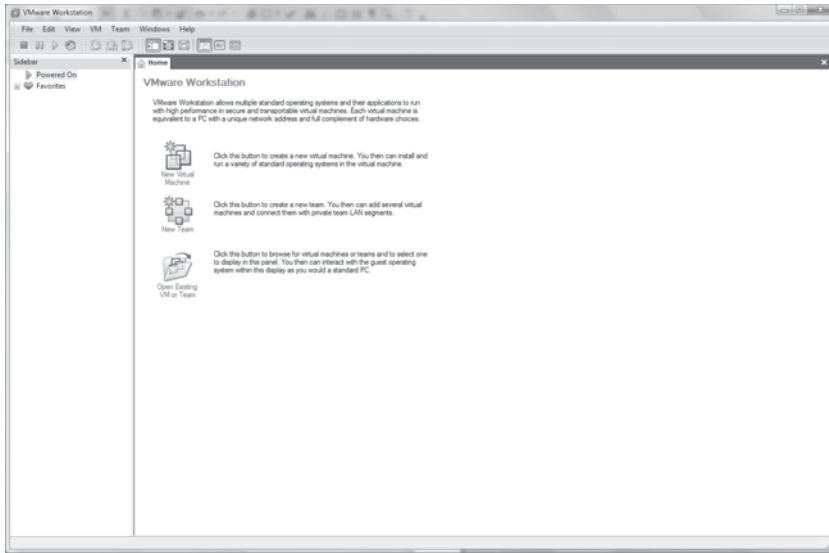


Figure 1-61

The VMWare desktop is the starting point of virtualization. It is the control center from which “virtual machines” are created to contain guest operating systems. To simulate an OS, VMWare requires users to create a virtual machine (an emulated computer), and there generally must be as many virtual machines as there are guest operating systems — one virtual machine per guest OS. The following section illustrates how to emulate Linux Ubuntu (guest OS) using VMWare running on the host OS Windows Vista, although these steps could equally apply to Windows XP as the host.

1.4.2 Creating a Virtual Machine for Linux Ubuntu

1. Start VMWare Workstation.
2. Select **File | New | Virtual Machine**.
3. The Virtual Machine Wizard appears. Click **Next**.



Figure 1-62

4. Select **Typical** and click **Next**.

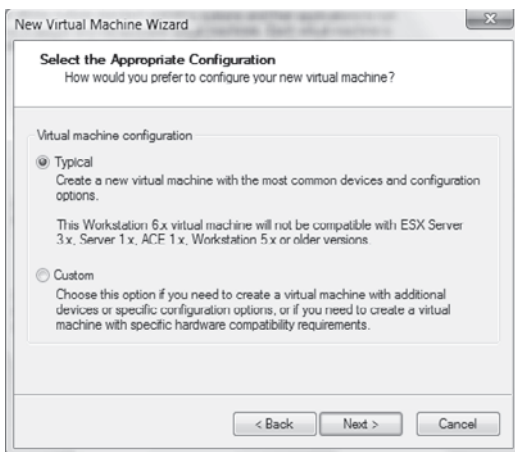
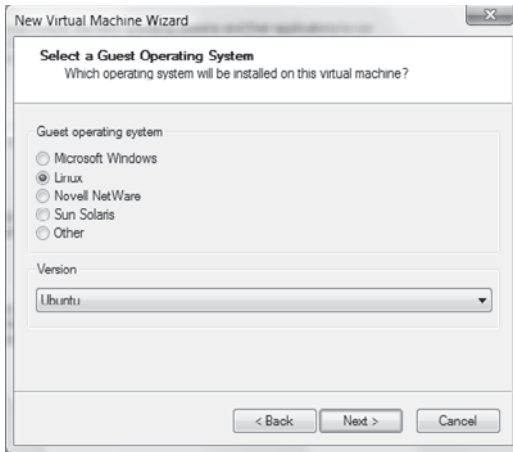
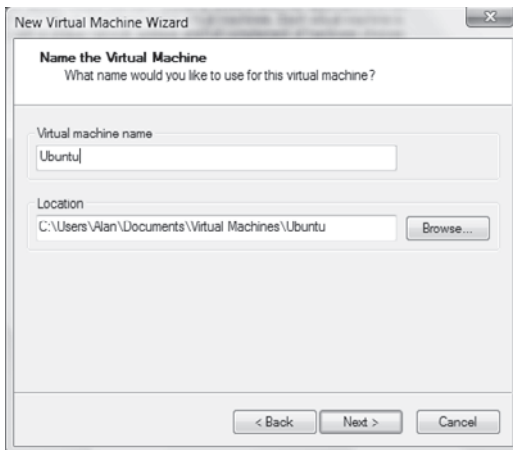


Figure 1-63

5. Select the guest OS to be run on this virtual machine — Linux for this example — and select Ubuntu from the drop-down list. Click **Next**.

*Figure 1-64*

6. Enter a name for the virtual machine, and select a folder on the host hard disk where both the virtual machine and guest OS will be “housed.” This will be “virtualized” as the hard disk (or virtual disk) for the guest OS. Click **Next**.

*Figure 1-65*

7. Select the desired networking option; here we've chosen Use network address translation (NAT), which allows the guest OS to connect to the Internet through the host connection. Click **Next**.

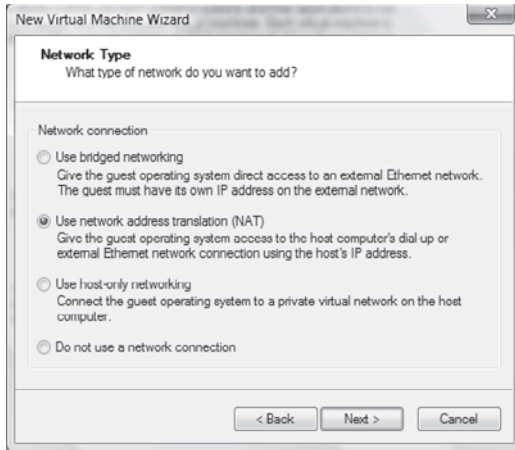


Figure 1-66

8. Specify the total amount of space in megabytes to be deducted from the host hard disk and used as the total capacity for the virtual disk of this virtual machine. For a virtual machine running Ubuntu, this capacity must be more than 2 GB (2,000 MB). Click **Finish**.

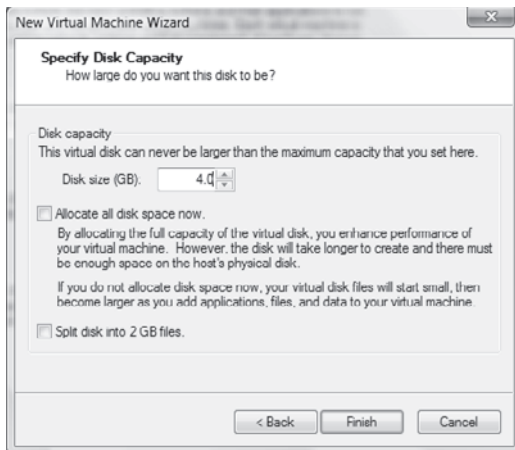


Figure 1-67

9. The virtual machine is created successfully and has a clean, formatted hard disk of a total capacity as specified in the previous step, and is now ready to install the guest OS, Ubuntu. Click **Close**.
10. Download a Linux Ubuntu ISO from the Ubuntu web site (<http://www.ubuntu.com/>), and then return to the VMWare desktop.
11. The VMWare desktop lists the newly created virtual machine, and the Devices panel lists the virtual machine's hardware (which can be customized). Double-click the CD-ROM device.

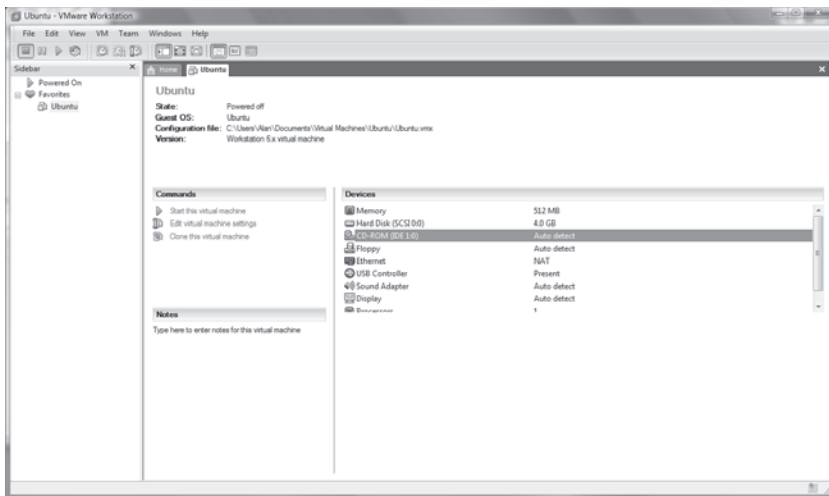


Figure 1-68

12. The CD-ROM device for the guest OS is currently mapped by default to the CD-ROM of the host OS, meaning each OS will share the CD-ROM. This CD-ROM can also be assigned to ISO images. To install Ubuntu to this virtual machine, choose **Use ISO image** and use the Browse button to select the downloaded Ubuntu ISO. Click **OK**.

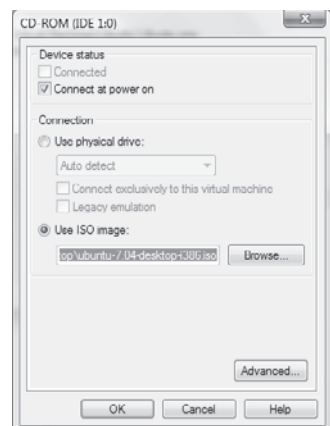


Figure 1-69

13. Select the **Ubuntu** virtual machine from the Sidebar and either click the **Play** button, or right-click the mouse on the Ubuntu icon and then click **Power-On** to start the virtual machine, automatically booting from the Ubuntu ISO.

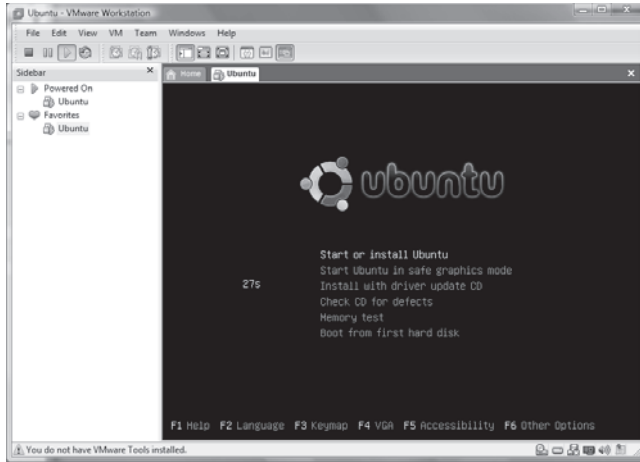


Figure 1-70

14. Install Ubuntu using the instructions from Section 1.3.5.
15. The guest OS is now installed on the virtual machine. Shut down the guest OS as usual to power-off the virtual machine, and restore the CD-ROM mapping settings from the Ubuntu ISO to the local machine's hardware, as discussed in step 12. The virtual machine can now be powered on to boot Ubuntu as a guest OS.

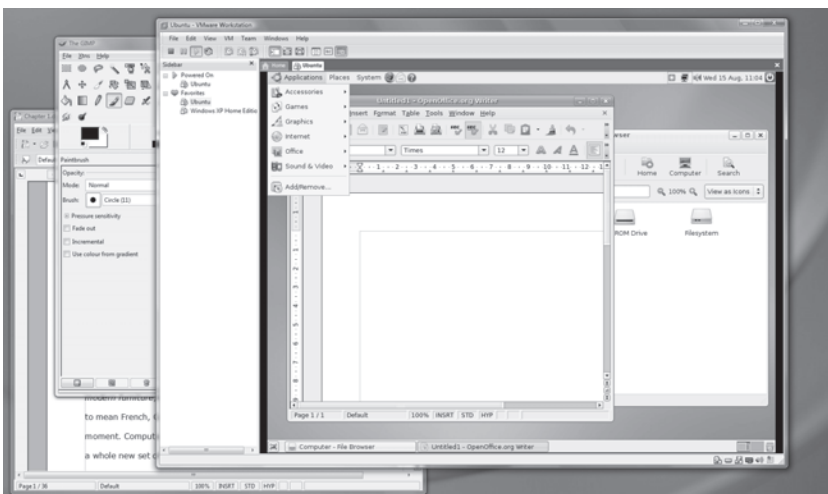


Figure 1-71

1.5 Conclusion

This chapter defined cross-platform games as games that run on two or more different platforms (or species of operating systems). Examples of cross-platform games include Diner Dash, The Battle for Wesnoth, OpenArena, and thousands of others. Some of the potential operating systems games can run on include Windows, Mac OS X, and Linux; Linux Ubuntu especially has attracted a growing population of game players and developers compared to other Linux distributions. To develop cross-platform games, developers must compile and test their games on each target platform before release as a final product. Thus, cross-platform development requires a cross-platform environment, and developers must have the facilities to run multiple platforms by maintaining one machine per platform, by multiple booting, or by using virtualization software like VMWare to simulate guest operating systems inside a host OS. This book assumes the reader has a Windows or Mac OS X background, and for this reason, Linux Ubuntu, and using Linux generally, is considered in more detail in the next chapter.

Chapter 2

Linux Ubuntu and Linux Gaming

In the previous chapter we discussed the various OS platforms commonly used for games, as well as the theories behind cross-platform games and their programming. We also discussed various machine configurations required in order to program and test these games, including multiple boot systems and OS virtualization.

This chapter moves a step forward in the world of cross-platform game development and puts behind us the technicalities of machine setup, multiple booting, and virtualization by considering further the nature of the Linux distro Ubuntu, and specifically the relationship between Linux and contemporary games. In short, this chapter is primarily a quick-start guide to Ubuntu and Ubuntu gaming, aimed largely at those users with only a Windows or Mac background who would like to learn more about Linux and Linux games.

2.1 Ubuntu Installation and Troubleshooting

In the previous chapter we discussed Linux Ubuntu, a Linux distribution, detailing what it is, how to download it, and how it can be installed to the system using the Ubuntu Live CD, freely available from Canonical, Ltd. (<http://www.ubuntu.com>). Ubuntu, like the increasingly popular PCLinuxOS, is a desktop Linux distribution intended for general home and multimedia use, such as browsing the net, watching movies, listening to music, and playing games. With periodical software updates and patches, biannual new releases, integration with Windows partitions and networks, multimedia codecs, a growing online community, increasing support for various hardware and drivers, and compatibility with many common Windows file formats (from Microsoft Office to Adobe Photoshop), Ubuntu looks to become one of the prominent Linux gaming platforms.

The previous chapter explained the details of a standard Ubuntu installation. Whether or not this install is for single OS machines, multiple boot machines, or virtualization configurations, the installation is the same for each. This section considers in more detail some of the problems encountered by users during an Ubuntu install, and provides some potential solutions and advice in a typical Q&A (questions and answers) format.

Q. I insert the Ubuntu CD/DVD and then restart the computer with the CD/DVD in the drive, but at system boot-up nothing happens. The system either boots into the OS installed on the machine already (Windows XP, Vista, etc.), or the machine does nothing but show a prompt or a blank screen, possibly because no OS is installed at all. It acts no differently from when it has an empty CD/DVD drive.

A. This problem occurs most likely because the system's BIOS (basic input/output system) is not configured to boot from the CD/DVD at system startup, or because the system has more than one connected CD/DVD drive and it is one of the other drives (not this one) from which it is configured to boot. If the latter, then this

issue is usually resolved by simply inserting the Ubuntu CD/DVD into a different drive and then rebooting the system; if, however, the former is the case (or if the “changing drives” solution doesn’t work), then the BIOS should probably be reconfigured to boot first from the CD/DVD and then subsequently from other bootable system devices, such as the hard disk, USB sticks, etc. The menus and options differ from one BIOS to another, and readers are advised to consult their BIOS manual or their computer supplier or hardware manufacturer before editing BIOS settings.

Q. I have already installed Windows Vista, and I have yet to install Ubuntu for a dual-boot arrangement. However, Windows Vista is installed to the primary partition on the system and there is only one partition, or there exist only NTFS formatted partitions. Is there any way I can install Ubuntu to the system without having to format the entire hard disk, including Vista, and start again? I would like to keep my existing Windows installation and simply “add” a new partition onto which I can install Ubuntu.

A. Yes, Windows Vista ships with a Disk Management Utility (accessible from My Computer | Manage | Disk Management), and this is designed to “shrink” (or cut off) parts of an existing or preformatted drive into a separate partition on-the-fly. Simply select a drive from which space may be deducted to feed the new partition, then click Shrink.

Q. During the Ubuntu installation, the Partition Wizard shows me a list of all disks and partitions to which Ubuntu may be installed; however, a selection of one or more small-sized disks appear, perhaps 256 MB in size, or maybe 512 MB, etc., and these devices are often given system-like names like `/dev/sdb1` or `/dev/sda3`. What are these?

A. These devices are probably USB memory sticks, TV tuner USB devices, or photographic cards; if this the case, then each of them can safely be ignored.

Q. During the Ubuntu installation, the Partition Wizard doesn’t display any available hard disks or partitions onto which Ubuntu can be installed.

A. This issue may be related to software RAID devices; this is perhaps solvable by using the Ubuntu Alternate Installation CD, downloadable from the Ubuntu web site. This CD does not include a Live version of Ubuntu, but instead features a text interface installer with a number of extra utilities designed to make installation simpler on “problematic” machines.

Q. After installing Ubuntu, I can no longer boot into Windows Vista, perhaps because Vista is no longer a selectable option from the boot menu at system startup.

A. Insert the Windows Vista DVD into the drive and reboot the computer. Booting from the DVD, enter the Vista Setup and select the Repair installation. Highlight the Vista partition and run the command prompt. Then enter the following command to run the Microsoft Chkdsk application and repair errors:

```
chkdsk c: /R
```

2.2 Getting to Know Ubuntu

Ubuntu is a completely free of charge, open-source Linux-based operating system (distro) for desktop computing (home use). Thus, it comes with bundled software for playing movies, listening to music, browsing the Internet, editing photos, performing office tasks like word processing, and of course, playing games. This section offers a broad overview of many applications preinstalled with Ubuntu, but cannot hope to offer a complete and comprehensive guide to Ubuntu. This chapter should be considered more as a “getting started” kit.

2.2.1 Ubuntu Login



Figure 2-1: Ubuntu login screen

Ubuntu — like some Windows and Mac systems — begins at the login screen where users may type their user names and passwords to enter their respective “areas” on the system. This is designed to authenticate user access to the computer, and to spatially divide the user-created documents on a per-user basis for personal photos, e-mail, documents, etc.

2.2.2 Ubuntu Desktop

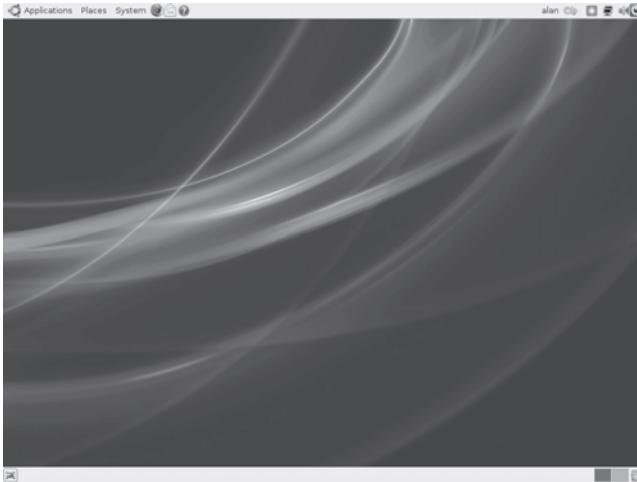


Figure 2-2: The Ubuntu desktop

Once a user logs into Ubuntu from the login screen, the OS then boots to the “desktop” (like the Windows or Mac desktop). From here users launch applications to browse the web, check e-mail, create documents, play games, and more. By default in Ubuntu, the main menu (the place from where applications are launched) appears as a gray bar aligned horizontally across the top of the screen, and it features three menu items: Applications, Places, and System. Applications is a menu of installed and executable applications, Places is a menu of shortcuts to common system locations such as the desktop, documents, and system devices, and System is a menu of control panel-like applications and other system management utilities.

2.2.3 System Monitor

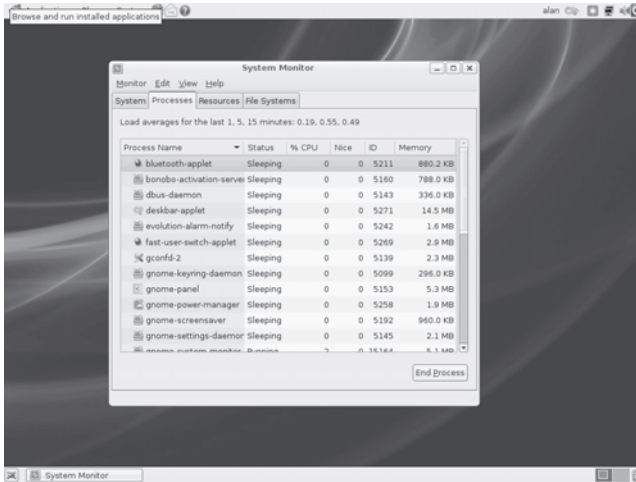


Figure 2-3: Ubuntu System Monitor

System Monitor, which is similar to the Windows Task Manager, is a utility application designed to display a list of all the currently running system processes (from applications to libraries and other invisible routines). It can be launched from the System menu by choosing System | System Monitor.

2.2.4 Update Manager

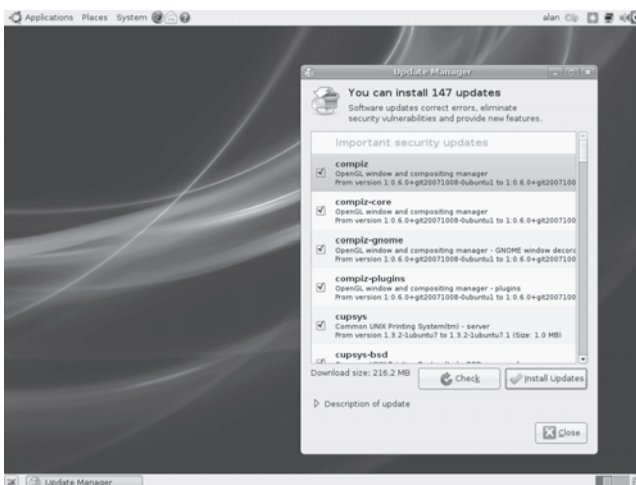


Figure 2-4: Ubuntu Update Manager

Both Windows and Mac have their own intrinsic automated patching or update services designed to update the installed applications and to repair known software bugs, from trivialities to security updates (e.g., Windows Update). Ubuntu also features an automated, online patching service that first scans the local computer for all installed applications and then notifies the users whenever appropriate updates or fixes are available for download. Update Manager is accessed by double-clicking the icon in the top right-hand corner of the screen.

2.2.5 Screen and Graphics Preferences and Restricted Drivers Manager

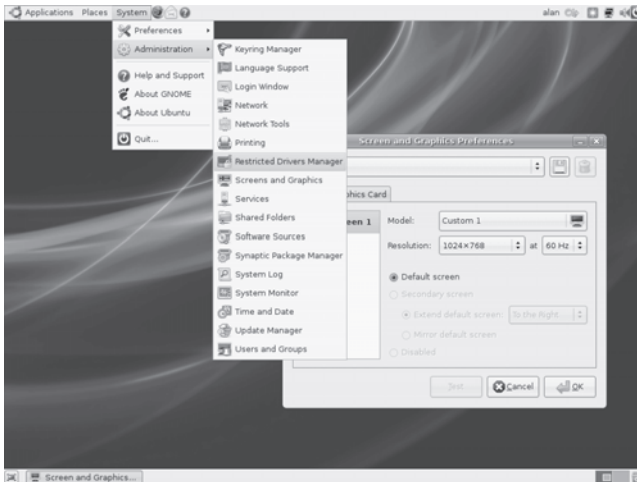


Figure 2-5: Ubuntu Screen and Graphics Preferences

The Screen and Graphics Preferences utility allows you to change the system resolution and the monitor type and to install device drivers for graphics cards. This utility can be accessed from the Ubuntu menu bar by selecting System | Administration Screens and Graphics. The Restricted Drivers Manager is also available from the System | Administration menu, and is designed to auto-detect graphics hardware (such as Nvidia and ATI hardware-accelerated cards) and install the appropriate drivers for them.

2.2.6 Add/Remove Applications

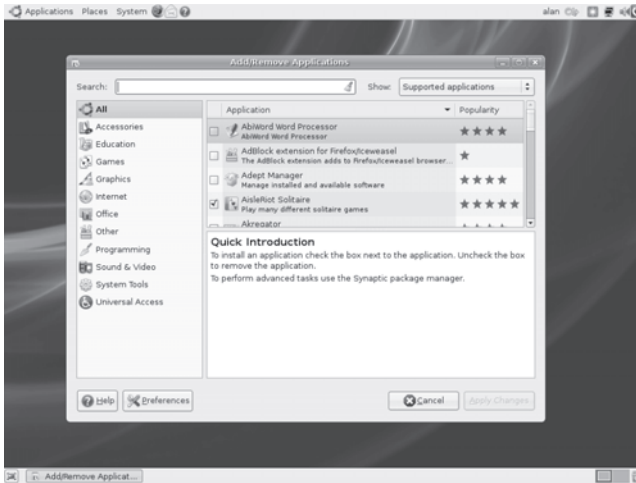


Figure 2-6: Ubuntu Add/Remove Applications

In Windows, the Add/Remove control panel utility is used primarily to remove (or uninstall) software from the system, and only secondarily to install software since most software ships with its own installers, typically install wizards. Not so in Ubuntu, however. In Ubuntu, the Add/Remove utility (accessible from the menu bar by selecting Applications | Add/Remove) is a comprehensive and actively maintained database of applications available to install free of charge. Installed applications may also be removed from the system using this utility.

2.2.7 Synaptic Package Manager

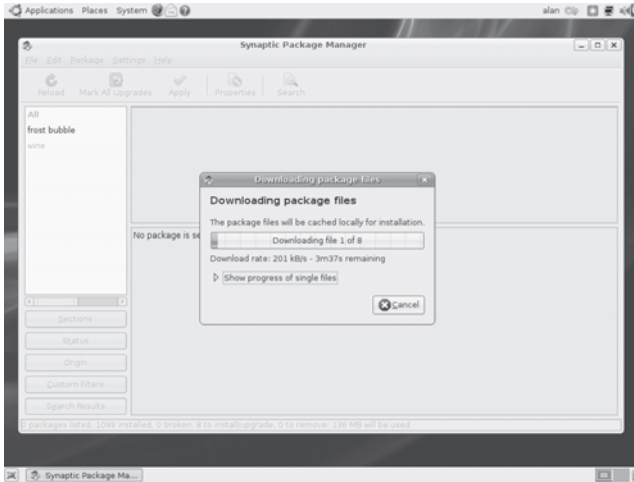


Figure 2-7: Downloading a file with Synaptic Package Manager

Like an advanced Add/Remove utility, the Synaptic Package Manager can do everything Add/Remove can do, and more. Accessible by selecting System | Administration | Synaptic Package Manager, the Synaptic Package Manager is a comprehensive database of both applications *and* non-executable software like libraries and software development kits (SDKs). This utility will be important for the Ubuntu game developer, as we shall see.

2.2.8 Ubuntu Terminal/Console/Shell



Figure 2-8: Ubuntu Terminal

Ubuntu is said to be a Linux distribution, and like all distros, Ubuntu is based on the Linux kernel, which is the heart and soul of the operating system and contains the core foundational rudiments that actually make things work. This is a level at which there is no GUI; at this level the OS communicates in terms of Linux commands, a sight usually hidden from a user's eyes. However, the Ubuntu Terminal/Console/Shell is a portal through which the user may directly type commands and may control the OS through a keyboard-based shell language known as BASH (Born Again Shell). This chapter will look at some basic shell commands, and though the shell is an integral part of Linux, this book will not consider the shell in depth. In cases where the shell is required in this book, a step-by-step guide is used to highlight the exact commands to type into the terminal. Those looking to master Linux are advised to familiarize themselves with the Linux Terminal.

2.2.9 Places | Computer

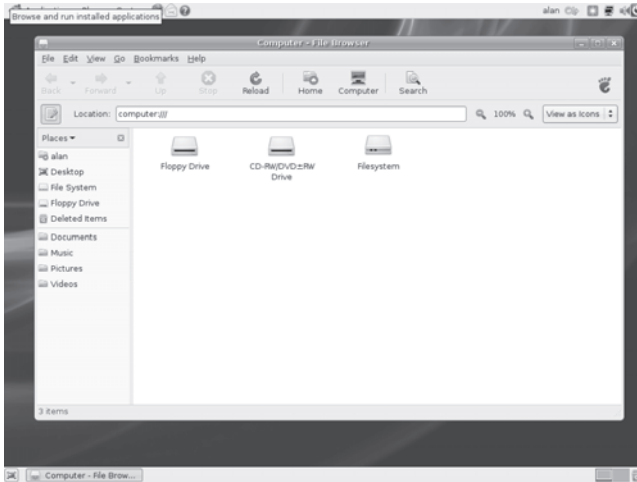


Figure 2-9

The Ubuntu equivalent of My Computer can be accessed from the main menu by selecting Places | Computer. From here, a user can browse all connected devices, internal and peripheral, from hard disks and CD/DVD drives to USB sticks and digital cameras.

2.2.10 Firefox Web Browser

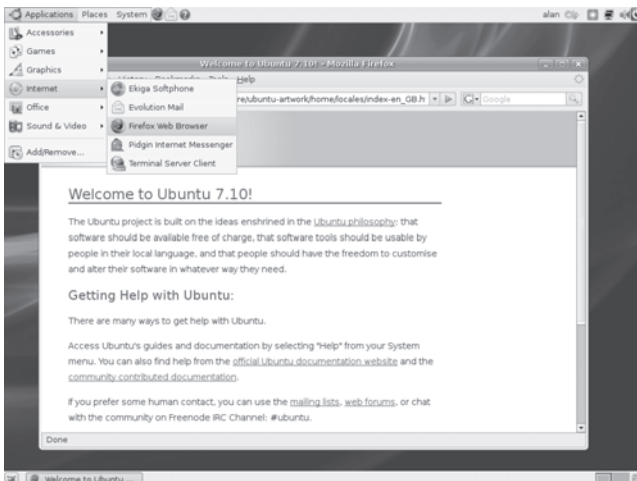


Figure 2-10: Firefox

Ubuntu ships by default with the open-source, cross-platform, and freely available web browser Mozilla Firefox. Like Internet Explorer, this application is used for browsing the web, checking e-mail, and also for browsing FTP servers. Firefox can be launched both from the shortcut icon and from the menu system itself via Applications | Internet | Firefox.

2.2.11 OpenOffice.org

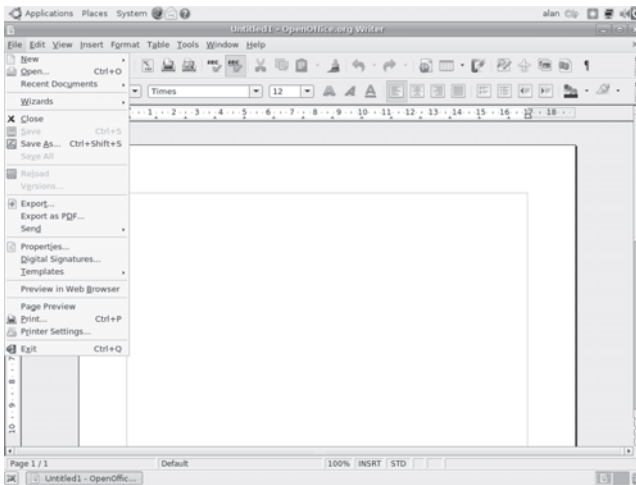


Figure 2-11:
OpenOffice.org

Ubuntu ships with the free, open-source, and cross-platform office suite OpenOffice.org, which is designed as a Microsoft Office alternative for creating documents, spreadsheets, animated presentations, and vector-based graphics. OpenOffice.org also features a celebrated degree of Microsoft Office compatibility, which means many documents and files can be migrated easily from Microsoft Office to OpenOffice.org.

2.2.12 Photo Editing

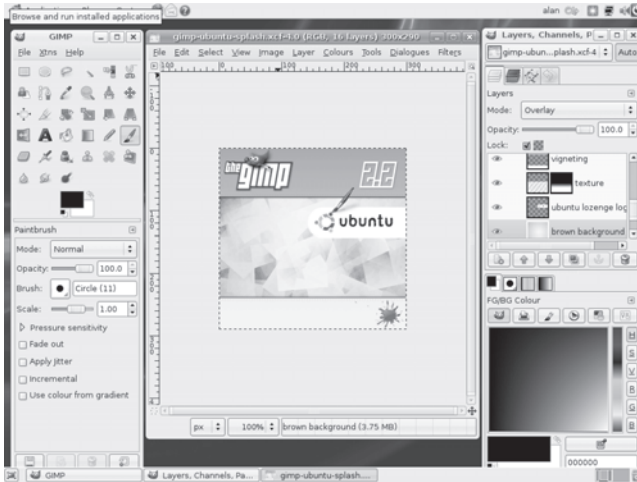


Figure 2-12: GIMP

A renowned open-source equivalent of Adobe Photoshop, GIMP is also shipped with Ubuntu and offers to developers a wide variety of tools for editing photos including brushes, stamps, crops, cuts, fills, and filters. GIMP is considered further as a cross-platform game development tool in the next chapter.

2.2.13 Installing and Playing a Game on Ubuntu



Figure 2-13: Frozen Bubble

Ubuntu is a promising gaming platform with a large selection of games available via the Add/Remove application database. This section provides a step-by-step guide to installing an open-source game called Frozen Bubble to the system via the Ubuntu Terminal console, demonstrating some shell commands and system tools along the way.



NOTE. Frozen Bubble is available as a web-based Java game (at <http://glenn.sanson.free.fr/v2/?select=fb:play>), but this tutorial focuses on the downloadable platform-specific binary distribution.

1. Beginning from the Ubuntu desktop, launch the Software Sources utility application by selecting **System | Administration | Software Sources**. This application expands the software database to include additional software sources.

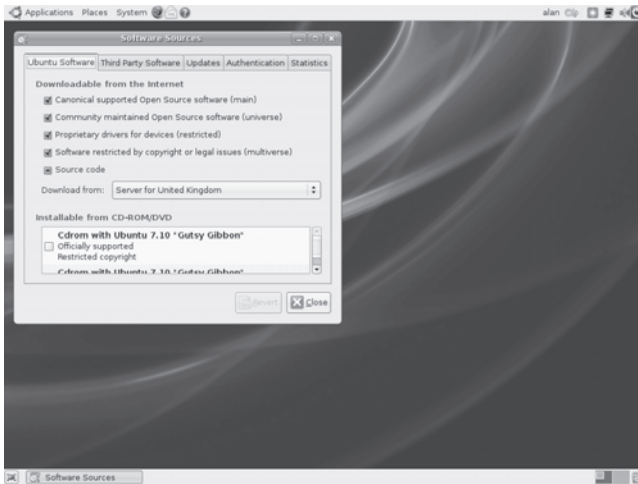


Figure 2-14

2. Check all software sources and then click **Close**.
3. From the Ubuntu main menu, open a terminal by selecting **Applications | Accessories | Terminal**.
4. Enter the following commands to install Frozen Bubble to the local machine, pressing the Return key after each line:

```
sudo apt-get update
sudo apt-get install frozen-bubble
```

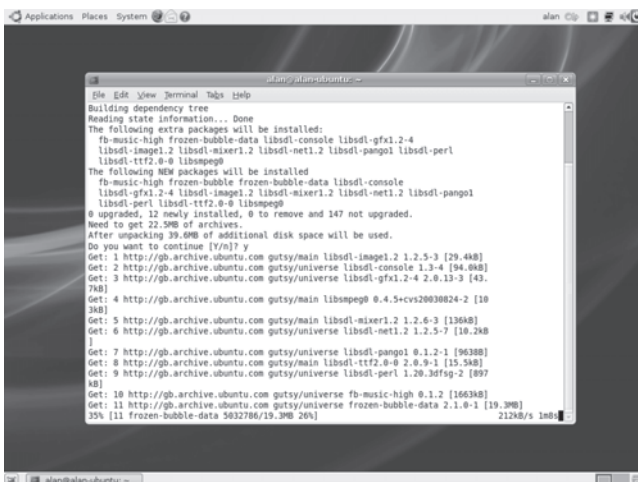


Figure 2-15

5. When the download is complete, close the terminal. Frozen Bubble is now ready to play via the Ubuntu main menu by selecting **Applications | Games | Frozen Bubble**.



Figure 2-16

2.3 Linux and “Transgaming” Technologies

One of the most common complaints leveled against Linux as a gaming platform is, “My whole gaming library is Windows based, and since most of my Windows games are not cross-platform, Linux can’t play my games.” Several years ago this argument was perhaps incontestable for the majority of Linux game enthusiasts. The point was that Linux couldn’t play Windows games, which was important for both gamers and developers. Important for gamers because gamers wanting to migrate to Linux would do so only so long as there was a way to continue playing their older Windows games, or to play recent games not available on Linux natively. Important for Linux game developers because without Windows gaming support on Linux, it would prove harder to attract a sizeable gaming population to the Linux platform. Transgaming applications (like Cedega and Wine), then, are seen by many as an attempt to resolve the platform boundaries facing gamers

on Linux; they are applications that allow Windows games to run on other platforms, including Linux. However, transgaming applications are *not* about cross-platform games. Why? Because cross-platform games are those that run *natively* on two or more different species of operating system, whereas transgaming technologies are a cross-platform compatibility layer supporting *platform-specific* games, allowing those games to run through the compatibility layer on other operating systems (hence the distinction in terms). Though this book is primarily about cross-platform games, this section examines a selection of compatibility layers (applications) that support transgaming on Linux. These applications allow Windows games to run on Linux, and specifically on Ubuntu.

2.3.1 Cedega

Cedega is a commercial cross-platform transgaming application, available for both Linux and Mac, that is designed for gamers to play Windows games on those other platforms. Some of the games supported by Cedega include Half-Life 2, Resident Evil 4, Need for Speed ProStreet, and Elder Scrolls IV: Oblivion. More details regarding Cedega can be found at the official Cedega web site at <http://www.transgaming.com/>.



NOTE. For a list of Cedega-compliant games, please visit the Cedega online game database at <http://games.cedega.com/gamesdb/>.

2.3.2 CrossOver

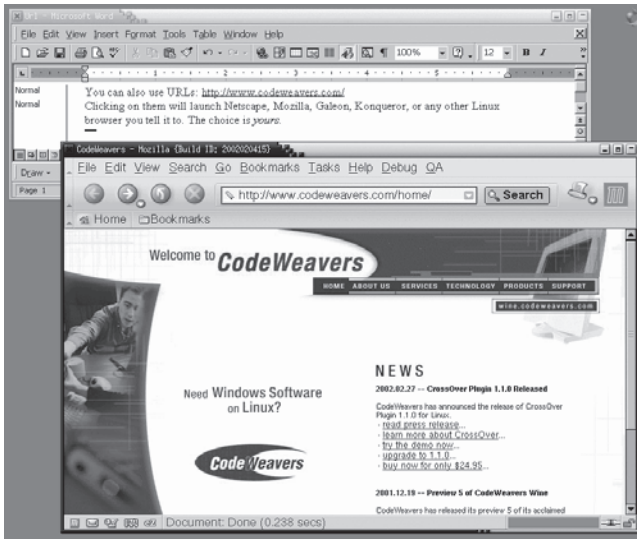


Figure 2-17: CrossOver

CrossOver is a commercial compatibility layer aimed less at supporting specifically Windows games on Linux than at supporting Windows applications generally on Linux and Mac (applications such as Microsoft Office, Photoshop, Director, etc.). However, CrossOver officially supports Shockwave Director for playing Shockwave-based web and stand-alone games. Other applications and games supported on Linux by CrossOver include Microsoft Office 2003, Photoshop 7.0, EVE Online, and others. For more details regarding CrossOver, please visit their web site at <http://www.codeweavers.com/products/>.



NOTE. For a list of CrossOver-compliant applications and games, please visit the CrossOver online games and applications database at <http://www.codeweavers.com/compatibility/browse/name/>.

2.3.3 Wine



Figure 2-18: Wine

Freely available, open-source, and cross-platform, Wine is a popular compatibility layer to run selected Microsoft Windows applications and games seamlessly on other platforms, such as Mac and Linux (running Windows applications as though they were native). Some of the Windows applications and games supported by the Wine application include Half-Life 2, .NET Framework 2.0, and World of Warcraft. More information can be found at the Wine web site at <http://www.winehq.org/>.



NOTE. For a list of Wine-compliant applications and games, please visit the Wine online games and applications database at <http://appdb.winehq.org/>.

2.3.3.1 Installing Wine on Linux Ubuntu

1. Beginning from the Ubuntu desktop, launch the Synaptic Package Manager from the Ubuntu main menu by selecting **System | Administration | Synaptic Package Manager**.

2. Click the **Search** button and search for **wine**.

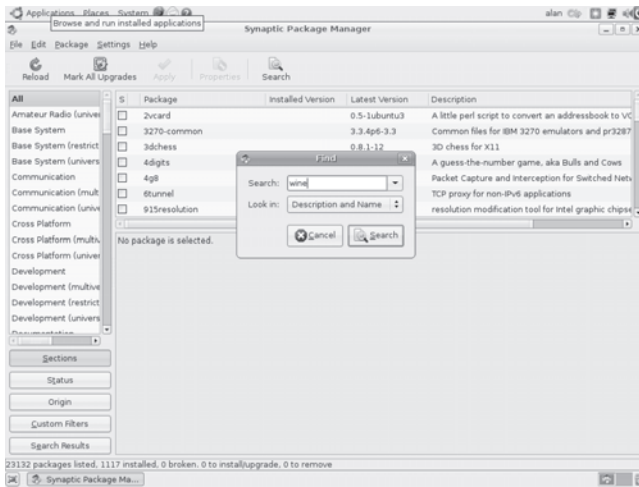


Figure 2-19

3. Click on the **wine** option that appears in the list to mark this software for installation, and then click **Apply** to install Wine. An Internet connection is required.

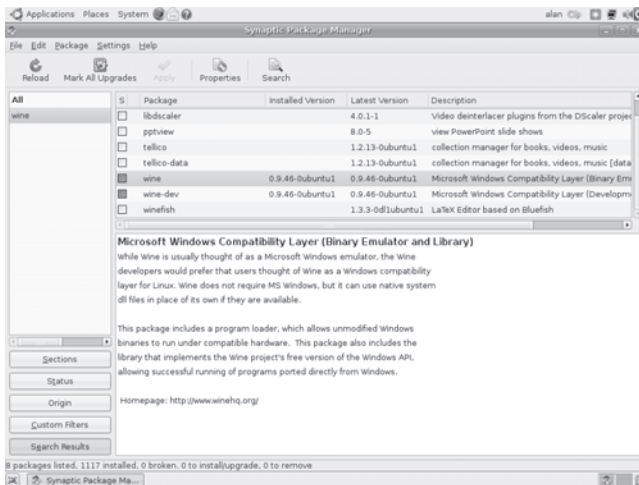


Figure 2-20

4. After the Wine application is installed, it is available from the Ubuntu main menu by selecting **Applications | Wine**.

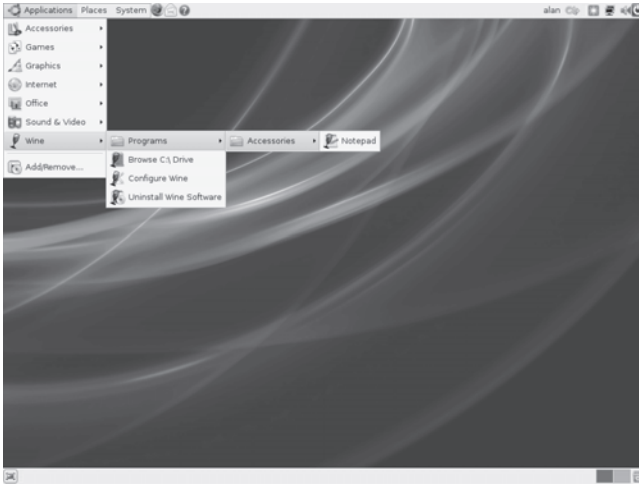


Figure 2-21

2.4 Automating Ubuntu with Automatrix

Ubuntu — like many Linux distributions — is free software (free of charge) but also free in terms of the FSF (Free Software Foundation) definition of free; as such it does not ship with certain media codecs and applications, specifically those that do not meet the FSF criteria for being “free.” Consequently, Ubuntu cannot *natively* play some specific media codecs that might be used in games such as WMA or MP3. There is, however, an Ubuntu application that may be downloaded for free and used to install a whole variety of popular media codecs and applications. The legal status of downloading specific codecs using Automatrix may vary from region to region across the United States and across the world. Readers are therefore advised to check the Automatrix documentation regarding its status in particular regions.



NOTE. The reader has sole responsibility for his or her downloads.

2.4.1 Installing and Using Automatrix for Linux Ubuntu

1. Beginning from the Ubuntu desktop, navigate the Firefox web browser to the Automatrix web site at <http://www.getautomatrix.com>.

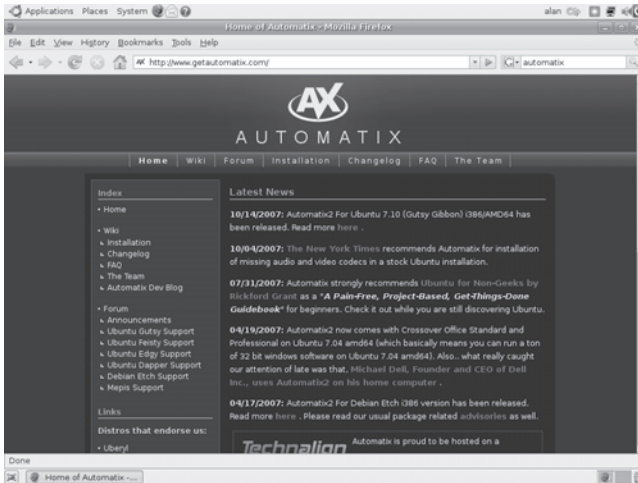


Figure 2-22

2. At the Automatrix home page, click the **Installation** link to display the installation page, and download the .deb installation package for your version of Ubuntu.



Figure 2-23

3. Once downloaded, double-click the .deb Automatrix installation package to install Automatrix to your local machine.

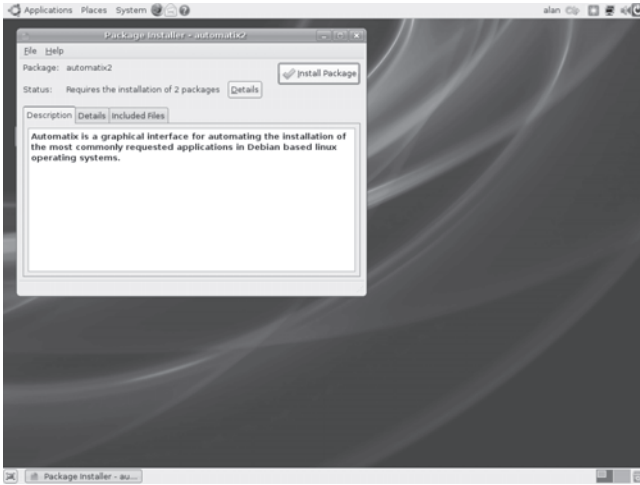
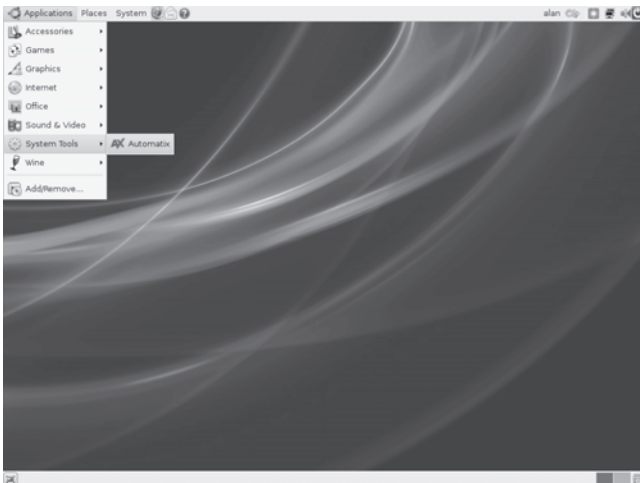


Figure 2-24

4. Launch Automatrix from the Ubuntu main menu by selecting **Applications | System Tools | Automatrix**.

Figure 2-25

- Once Automatrix is started, its menus can be used to select different products to install to the system. To begin the installation of checked items, click the **Start** button.

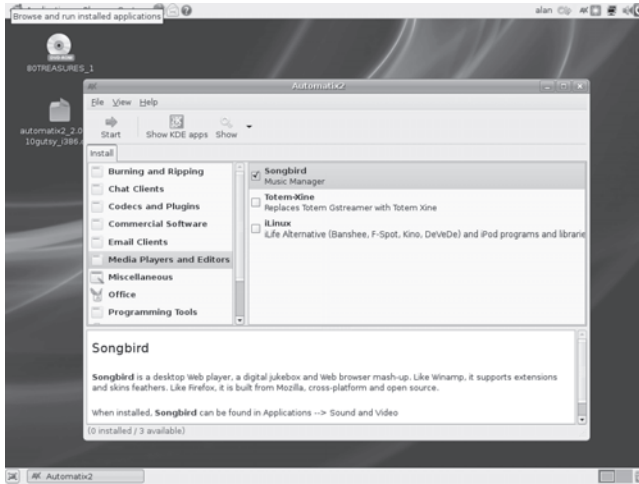


Figure 2-26

2.5 The Linux Shell

Ubuntu as a Linux distribution is based on the Linux kernel, which contains the OS components such as the foundational libraries and low-level hardware programming common to all Linux distributions. The kernel is not in itself a stand-alone OS insofar as users do not download and use the Linux kernel directly as an OS like they do a distro; instead, users download and use Linux-based distros (such as Ubuntu), themselves high-level layers that work “under the hood” with the kernel. The Linux kernel and the OS generally operate through a series of sending and receiving commands beneath the GUI. Users can send commands to the OS via the Ubuntu Terminal. This section considers further the Ubuntu Terminal and a selection of common BASH shell commands.



NOTE. At this point, open an Ubuntu Terminal from the main menu by selecting Applications | Accessories | Terminal.

2.5.1 Common Shell Commands

- The `ls` (list) command is perhaps the most common of all Linux BASH shell commands. Once entered into the Terminal window at the command prompt, `ls` returns in columns and rows an alphabetical list of the files in the current directory, as shown in the following figure.

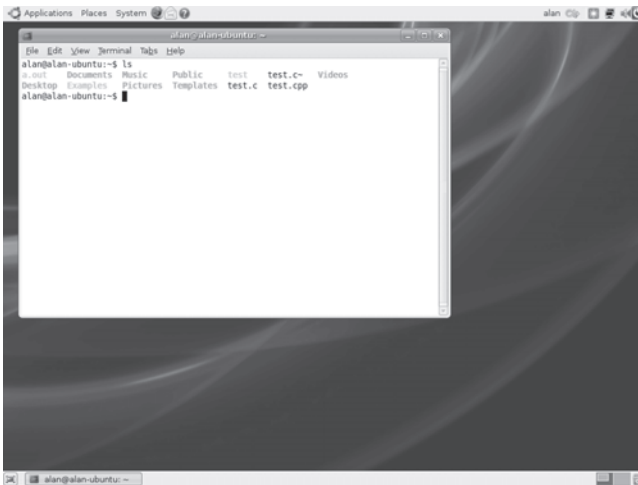


Figure 2-27

- The `clear` (clear screen) command clears (erases) all contents from the Terminal window.

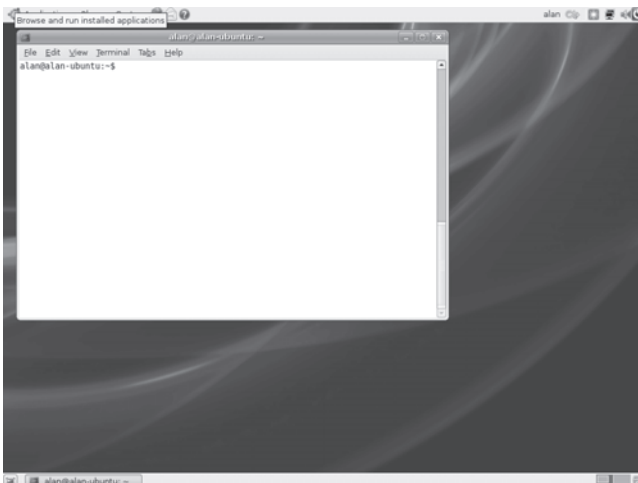
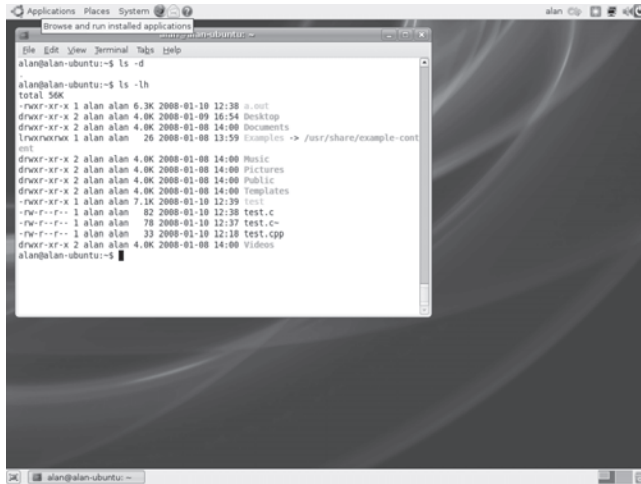


Figure 2-28

- Users may also add parameters to Linux commands. For example, “--help” displays on-screen help that details a command, its usage, and all possible parameters. Notice that parameters are passed to commands with a preceding hyphen (-) or double hyphen (--). The --help parameter applies to most Linux commands.



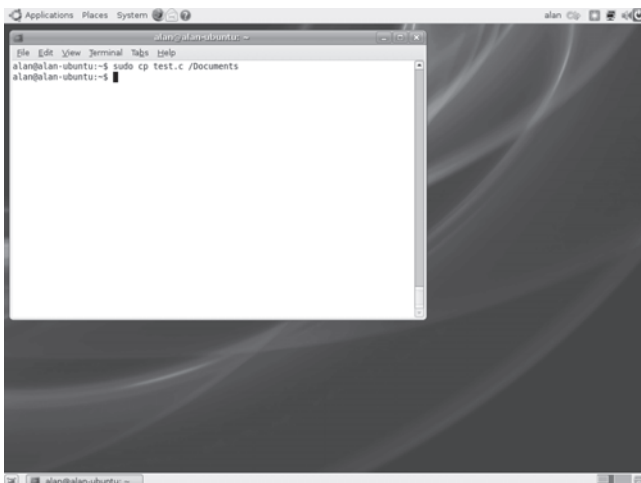
```

alan@alan-ubuntu:~$ ls -d
alan@alan-ubuntu:~$ ls -lh
total 50K
-rwxr-xr-x 1 alan alan 6.3K 2008-01-10 12:38 a.out
drwxr-xr-x 2 alan alan 4.0K 2008-01-09 16:54 Desktop
drwxr-xr-x 2 alan alan 4.0K 2008-01-08 14:00 Documents
lrwxrwxr-x 1 alan alan 26 2008-01-08 13:59 Examples -> /usr/share/example-cont
mit
drwxr-xr-x 2 alan alan 4.0K 2008-01-08 14:00 Music
drwxr-xr-x 2 alan alan 4.0K 2008-01-08 14:00 Pictures
drwxr-xr-x 2 alan alan 4.0K 2008-01-08 14:00 Public
drwxr-xr-x 2 alan alan 4.0K 2008-01-08 14:00 Templates
-rwxr-xr-x 1 alan alan 7.1K 2008-01-10 12:39 test
-rw-r--r-- 1 alan alan 82 2008-01-10 12:38 test.c
-rw-r--r-- 1 alan alan 79 2008-01-10 12:37 test.c.
-rw-r--r-- 1 alan alan 33 2008-01-10 12:18 test.cpp
drwxr-xr-x 2 alan alan 4.0K 2008-01-08 14:00 Videos
alan@alan-ubuntu:~$

```

Figure 2-29

- The cp (copy) command copies one or more specified files or folders to the specified destination folder. The command accepts two string arguments; the first specifies the path or file name of the source file or folder to copy, and the second specifies the destination to receive the copied file or folder.



```

alan@alan-ubuntu:~$ sudo cp test.c /Documents
alan@alan-ubuntu:~$

```

Figure 2-30



NOTE. To run commands as an Admin (superuser), the term “sudo” is prefixed to general system commands like cp.

- The cd (change directory) command is used to change the current directory to a specified subdirectory, and the command rm (remove direction) is used to remove (delete) specified file(s) from the current directory.

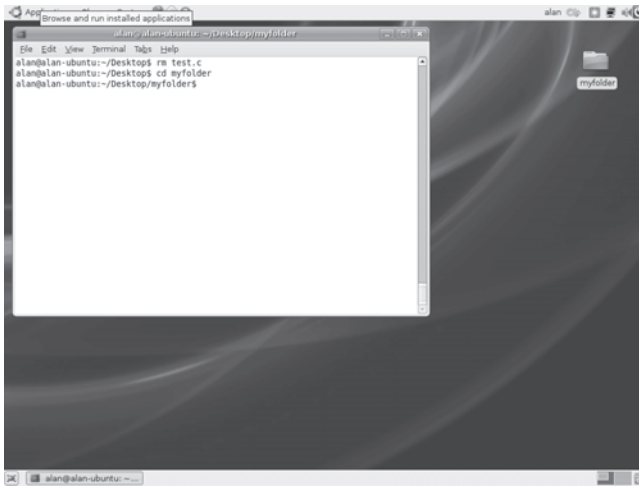


Figure 2-31



NOTE. File names featuring the space character, such as hello world.bmp, should be enclosed in quotation marks in the command line (“hello world.bmp”).

- The mkdir (make directory) command is used to create in the current directory a new directory of the specified name.
-

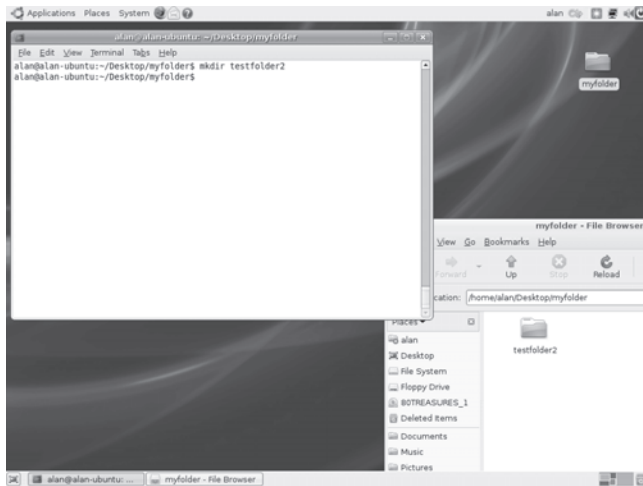


Figure 2-32

The following table lists a series of common Linux BASH shell commands, summarizing the role of each command and relating them to their DOS equivalent (for those readers familiar with DOS commands).

DOS Command	Linux BASH Command	Description
ASSIGN	ln	Create shortcut link to file or directory
CD	cd	Change directory
CHDISK	du -s	Check disk usage with summarize parameter
CLS	clear	Clear the screen
COMP	diff or cmp	List differences between the content of any two files
COPY	cp	Copy file or directory
DATE	date	Display system date
DEL	rm	Remove (delete) file
DIR	ls	List directory contents
DIR *.* /o-d	ls -tr	List directory contents by reverse time of modification/creation

DOS Command	Linux BASH Command	Description
DIR *.* /v /os	ls -ls	List files and size
ECHO	echo	Print variable to terminal window
EXIT	exit	Exit from terminal
HOSTNAME	hostname	Print to terminal window the host name of the computer
MD	mkdir	Make directory
MORE	more	Output contents of file to terminal page by page to fit the display
MOVE	mv	Move file
MSD	lsdev	Display system information
PING	ping	Check network connection
PRINT	lpr	Print text file
RD	rmdir	Remove directory
REBOOT	shutdown -r now	Shut down and reboot the machine immediately
SCANDISK	fsck	Check and repair the file system
SORT	sort	Sort data alphabetically or numerically
TIME	time	Display system time
TREE	ls -r	List files in reverse order
TYPE	cat	Output contents of file to terminal
WIN	startx	Start X server
XCOPY	cp -R	Copy directory and all files and subdirectories recursively



TIP. Comprehensive guides and tutorials to the Linux shell are freely available online and can be found at the following web sites:

- <http://www.usd.edu/~sweidner/lst/>
 - http://www.linuxcommand.org/learning_the_shell.php
 - <http://www.gnu.org/software/bash/manual/bashref.html>
-

2.5.2 Creating and Compiling a C Program Using the Ubuntu Terminal and BASH Shell Commands

Computer games are often made in languages such as C and C++, regardless of whether the game runs on Linux, Windows, or Mac. In Ubuntu, and Linux more generally, C programs can be created and compiled entirely via the shell. However, most game developers nowadays use an IDE (integrated development environment) instead of the terminal. Some IDEs are slick-looking editors complete with compiler, debugger, code editor, and a series of other features to make programming simpler. A cross-platform code editor, Code::Blocks, is considered in the next chapter. The following steps illustrate how to create, compile, and run a program via the Ubuntu Terminal using only the keyboard and the BASH shell commands.

1. Beginning from the Ubuntu desktop, launch a Terminal window from the Ubuntu main menu by selecting **Applications | Accessories | Terminal**.
2. Enter the desktop folder by typing the following command into the Terminal window:

```
cd desktop
```

3. From the desktop folder, use the BASH shell commands to create a source file called `test.c` (a text file to contain the C source code for a simple C program), as follows:

```
sudo gedit test.c
```

4. A text editor application begins. Enter into the text editor the following simple C program that will print “hello world” into the Terminal window whenever it is executed from the terminal:

```
#include <stdio.h>
int main()
{
    int i;
    for (i = 0; i < 10; i++);
    printf("hello world\n");
}
```

```
    return 0;  
}
```

5. Once this is entered into the editor, select **File | Save** to save the file locally. Then exit the gedit program and return to the terminal.
6. At the prompt, enter the following to “dump” to the terminal the contents of the C file, and ensure the contents are correct.

```
cat test.c
```

7. Compile the C file using the following Terminal command:

```
gcc -ggdb test.c -o test
```

8. Once compiled successfully, run the newly compiled program using the following Terminal command:

```
./test
```

2.6 Conclusion

This chapter has highlighted both the intricacies of Ubuntu as an operating system, and how Ubuntu — being lightweight, free, and versatile — is likely to become an important Linux distribution for gamers and game developers.

The next step is to examine a selection of cross-platform game development tools such as a C++ programming IDE and a couple of graphics suites. Specifically, the next chapter examines Code::Blocks, GIMP, Blender 3D, and Audacity, which can be considered cross-platform tools insofar as each of them run on Windows, Mac, and Linux and each of them can be used to create games for all three of those platforms.

Chapter 3

Cross-Platform Development Tools

The first chapter of this book considered the definition of “cross-platform” and examined a variety of contemporary operating systems from Windows to Linux. The second chapter offered a beginner’s overview of Linux Ubuntu, focusing specifically on Ubuntu gaming and game development. This chapter is guided by the ethos that “a workman is only as good as his tools,” and so it takes the first step along the twisted road of game development by selecting (and installing) all the necessary cross-platform tools and software for making cross-platform games. Hundreds of open-source applications are available to developers for the purpose of making cross-platform games, downloadable and free of charge to anyone with an Internet connection. We’ll discuss four of these applications in this chapter: Code::Blocks, GIMP, Blender 3D, and Audacity.

- **Code::Blocks** — First released in 2004, Code::Blocks is an open-source and free software IDE (integrated development environment) for C++, supporting a selection of platforms including Windows, Linux, FreeBSD, and Mac OS X. Sporting a host of features including project management, syntax highlighting, and code completion, Code::Blocks allows programmers to compile C++ applications on all of its supported platforms. This chapter examines how to download, install, and use Code::Blocks on Windows and Linux, although it assumes the reader is already familiar with C++ as a language.

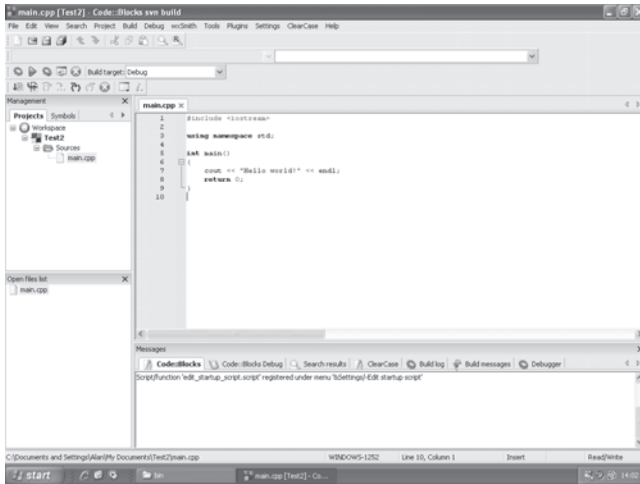


Figure 3-1:
Code::Blocks

- **GIMP** — An alternative to Adobe Photoshop, GIMP is an open-source and free software photo editing suite with a list of features almost as long as its developmental history. GIMP offers to its users a varied selection of tools ranging from brushes to highlighters designed specifically for retouching, scanning, and editing photos and textures.



Figure 3-2: GIMP

- **Blender 3D** — Blender 3D as an application was born in 1998, and has since 2002 developed under the guidance of Ton Roosendaal to become the world's most downloaded 3D animation program, with the online encyclopedia Wikipedia estimating Blender's user base at over 800,000 worldwide. Blender 3D features a competitive toolset for making 3D animations, models, and other special effects for both movies and games. Unlike its proprietary competitors such as 3ds Max, Maya, and SoftImage, Blender 3D is free and open-source, and has further gained fame from its association with movies such as *Spider-Man 2*, the short film *Elephants Dream*, and the Argentine CG movie *Plumíferos*. Blender 3D supports the Windows, Mac OS X, Linux, SGI, Irix 6.5, and Sun Solaris 2.8 platforms.



Figure 3-3: A render from Blender 3D

- **Audacity** — The digital audio editor Audacity is designed for recording, editing, and exporting audio ranging from less than a second in length to full soundtracks. Audacity was awarded the 2007 Community Choice Award for Best Project for Multimedia by SourceForge.net, and in August 2007 Audacity was ranked by SourceForge.net as its 11th most popular app, having been downloaded 24 million times. Furthermore, like the other software featured in this chapter, Audacity is open-source, free, and cross-platform, and supports Mac OS X, Microsoft Windows, and Linux. Audacity is discussed in more detail in Chapter 6.

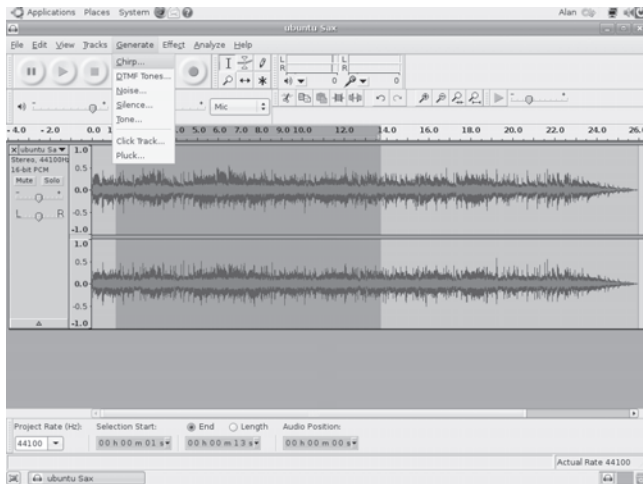


Figure 3-4:
Audacity

3.1 Code::Blocks

Games are only one form of software as are word processors, spreadsheets, databases, and Internet browsers like Internet Explorer or Firefox. Regardless of the kind of software, however, all software comes to exist only after having first been developed by programmers using a programming language such as C++. When developing their software, most programmers make use of other kinds of software to make their professional lives easier. Code::Blocks is one such package. Known as an IDE (integrated development environment), Code::Blocks is used by C++ programmers to manage and collate source files into organized projects, and to edit and compile source code into a final executable form that users can run stand-alone as an application. There are a variety of alternative development environments including Microsoft Visual Studio .NET and Dev C++; however, Code::Blocks is the focus of this chapter and is assumed to be the development environment used by the reader when considering all subsequent C++ code samples featured in this book. This is for the following reasons:

- Code::Blocks features a clean, approachable, and platform-independent interface that maps across-platforms, and so is equally usable on every supported platform.
- Code::Blocks features a comprehensive set of application wizards and project templates that can be used for every newly created project. These templates are specifically targeted toward game programmers beginning new projects in Code::Blocks, meaning cross-platform games may have their source code and foundations generated, configured, ready to execute, and ready to tweak within a few mouse clicks.
- Code::Blocks is open-source, free of charge, and freely downloadable for every user on any supported platform with access to the Internet.
- Being cross-platform means Code::Blocks is available for multiple platforms. Each build of Code::Blocks for a specific platform is known as a *distribution*, or *distro*, with one distribution for Windows, one for Linux, etc. Thus, projects created and compiled with one distribution of Code::Blocks may usually be migrated to and compiled by any other distribution (cross-compilation).
- Code::Blocks is a community-driven open-source project and, like so many open-source projects, has a thriving and supportive online community. Code::Blocks is regularly updated and widely documented, and the online forums offer a social environment for Code::Blocks enthusiasts from every platform.

3.2 Downloading and Installing Code::Blocks in Ubuntu

Code::Blocks is a free C++ IDE available to download for Linux Ubuntu from the official Code::Blocks web site or via the Ubuntu Terminal, but not from the Synaptic Package Manager like many other Ubuntu applications. The following steps detail the Code::Blocks installation process for Ubuntu.



TIP. Open-source applications are often considered by developers as works in progress, and Code::Blocks is no different. Developers often change them by adding new features and repairing existing bugs, and sometimes make their applications available in new ways. The following installation instructions for Code::Blocks on Ubuntu are known to be current at the time of writing, but readers may first prefer to check the online documentation at the Code::Blocks wiki (<http://wiki.codeblocks.org/>).

1. Beginning from the Ubuntu desktop on a machine with access to the Internet, open the Ubuntu Terminal by selecting **Applications | Accessories | Terminal**.

2. Type the following, pressing **Return** after each line:

```
sudo apt-get install build-essential
sudo apt-get install gdb
sudo apt-get install libwxgtk2.6-0
sudo apt-get install libwxgtk2.6-dev wx2.6-headers wx-common
sudo apt-get install wx2.6-doc
```

3. Back up the package sources file using the following command and then press **Return**:

```
sudo cp /etc/apt/sources.list /etc/apt/sources.list.bak
```

4. Edit the sources file as follows:

```
gksudo gedit /etc/apt/sources.list
```

Add the following line as appropriate for the particular Ubuntu distro, then save the document and return to the terminal:

```
(edgy main)
```

```
deb http://apt.tt-solutions.com/ubuntu/
```

or

(feisty main)

```
deb http://apt.tt-solutions.com/ubuntu/
```

5. Type the following, pressing **Return** after each line:

```
wget http://www.tt-solutions.com/vz/key.asc
sudo apt-key add key.asc
sudo apt-get update
sudo apt-get dist-upgrade
sudo apt-get install libwxgtk2.8-0 libwxgtk2.8-dev wx2.8-headers
wx-common
sudo update-alternatives --config wx-config
```

6. Exit the terminal and navigate from Ubuntu's web browser to the Code::Blocks web site at <http://www.codeblocks.org/>.

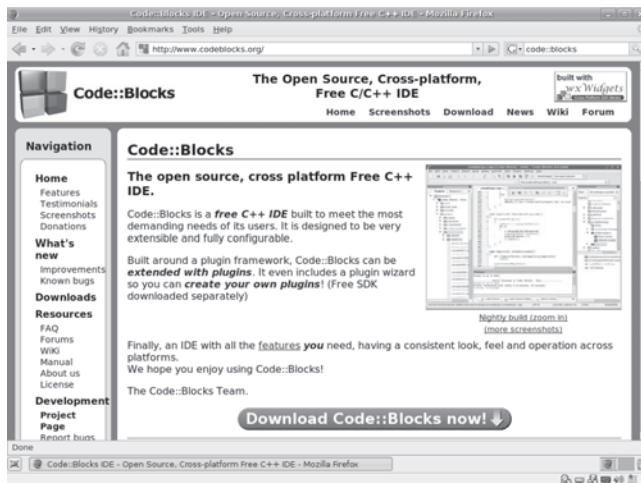


Figure 3-5:
Code::Blocks home
page

7. Click **Download** to navigate to the Code::Blocks download page, and from there select the latest Nightly Build option. Code::Blocks as an application may be downloaded from its official web site as one of two versions: a standard frozen release dating from 2005 (at the time of writing) or an experimental nightly “build,” typically released within the past 24 hours. The term “build” in this context refers to a complete and newly compiled version of Code::Blocks featuring any amendments and bug fixes coded by the

development team within the past 24 hours. For this reason, users are encouraged to download the latest nightly build of Code::Blocks rather than the frozen release since it boasts a greater variety of features and bug fixes. Choosing a nightly build from the web site sends users to the Nightly Build forum, a place where daily threads are arranged in date order with more recent dates toward the top, with each thread offering a link to the Code::Blocks build for the day the thread was posted.

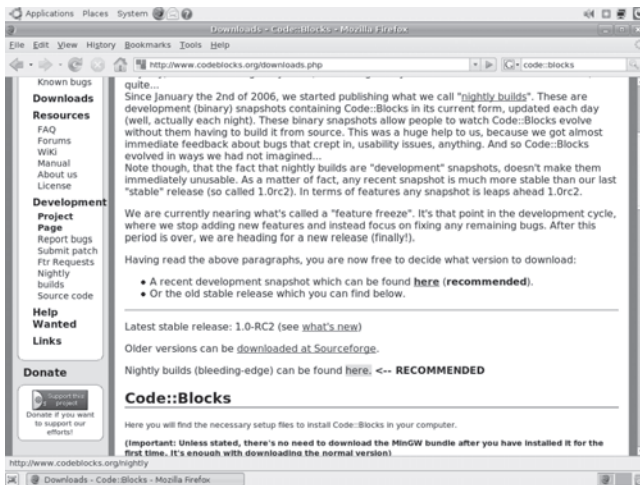


Figure 3-6:
Code::Blocks
download page

8. Select the latest nightly build thread available from the forum, and download the appropriate Ubuntu .deb package file to the Ubuntu desktop.

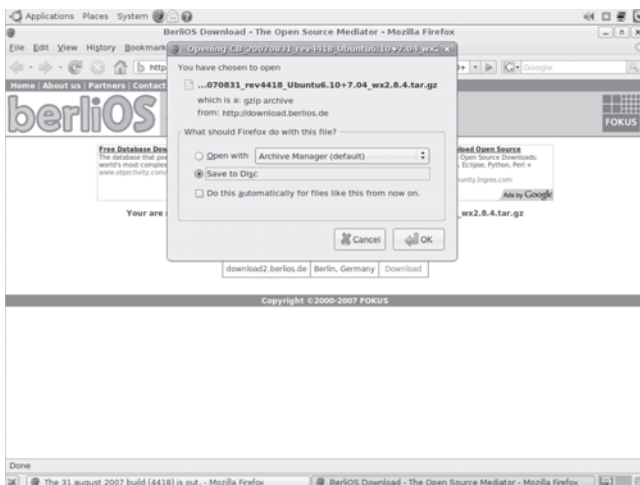


Figure 3-7

- Open the downloaded Code::Blocks archive set and extract all archives to the desktop, then proceed to install each archive to the system. Code::Blocks is now installed and is available from the Ubuntu Applications menu.

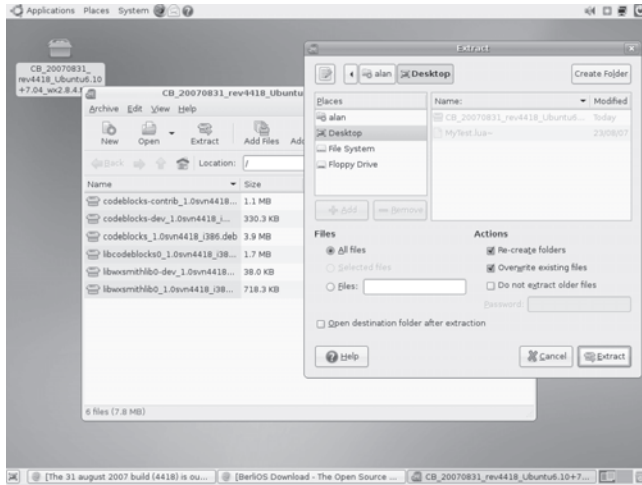


Figure 3-8

- From the Ubuntu main menu, choose **Applications | Programming | Code::Blocks** to launch Code::Blocks.

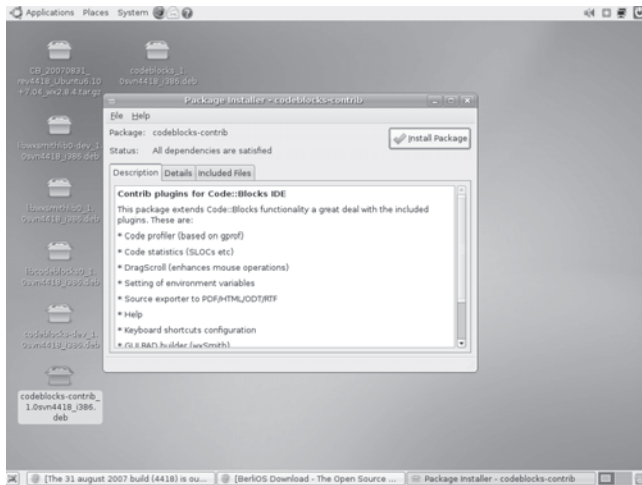


Figure 3-9

3.3 Downloading and Installing Code::Blocks in Windows

The following step-by-step instructions illustrate the Code::Blocks download and installation procedure for Windows.



TIP. Open-source applications are often considered by developers as works in progress, and Code::Blocks is no different. Developers often change them by adding new features and repairing existing bugs, and sometimes make their applications available in new ways. The following installation instructions for Code::Blocks on Windows are known to be current at the time of writing, but readers may first prefer to check the online documentation at the Code::Blocks wiki (<http://wiki.codeblocks.org/>).

1. Starting from the Windows desktop, launch a web browser and navigate to the 7-Zip Free Archiver home page at <http://www.7-zip.org/>. Choose a download and click the appropriate link. 7-Zip is a free, open-source file archiving application (like WinZip) for packing and extracting compressed archives in the following formats: 7Z, ZIP, GZIP, BZIP2, and TAR. This application is required because Code::Blocks is packaged and distributed to users in a series of 7Z archives.

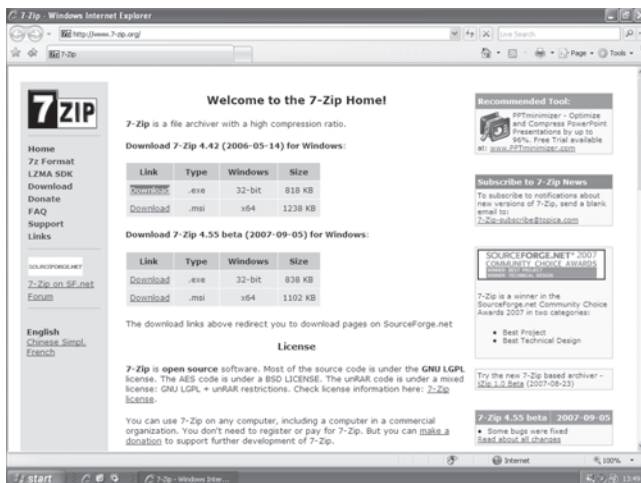


Figure 3-10:
7-Zip home page

2. Once 7-Zip is downloaded from the web successfully, execute its automated installer to install 7-Zip to the local computer.
3. Launch a web browser and navigate to the MinGW C++ Compiler web site at <http://www.mingw.org/>. Code::Blocks is compliant with a wide selection of popular C++ compilers but as an application it ships stand-alone, without a compiler, and therefore an independent compiler should be downloaded separately. MinGW is among several Code::Blocks-compatible packages, and features a suite of freely available, open-source C++ compilers and other tools for building C++ applications from standard C++ source and header files. To download the MinGW package from its official web site, select **Download | Installing MinGW**, and then select the automated installer.

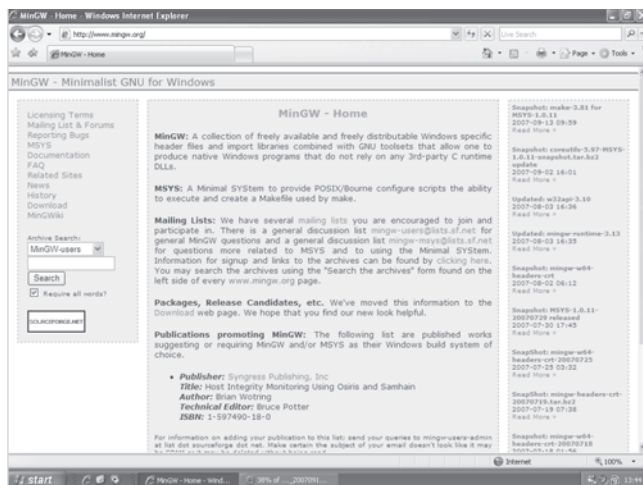


Figure 3-11:
MinGW home
page

4. Run the downloaded MinGW Automated Installer, and click **Next** to begin.



Figure 3-12:
MinGW download
dialog

5. Select **Download and install** and click **Next** to begin the online MinGW installation process, installing MinGW from the web to the local computer.
6. Once MinGW is installed to the local computer, launch a web browser and navigate to the Code::Blocks web site at <http://www.codeblocks.org>.

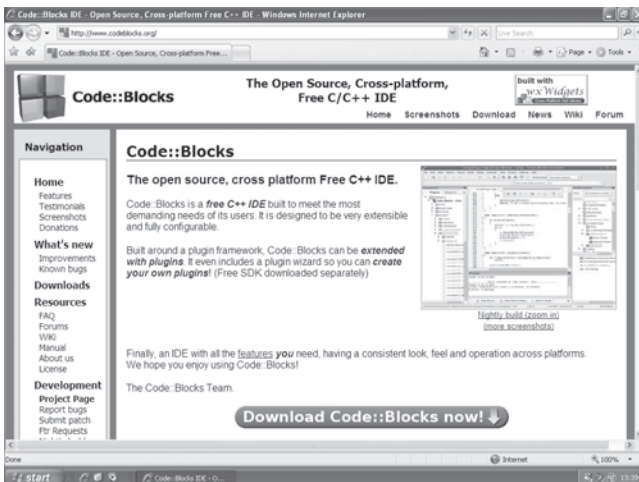


Figure 3-13

- Click **Download** to navigate to the Code::Blocks download page, and from there select the latest Nightly Build option. For more information on Nightly Builds generally, see step 7 of Section 3.2, “Downloading and Installing Code::Blocks in Ubuntu.”

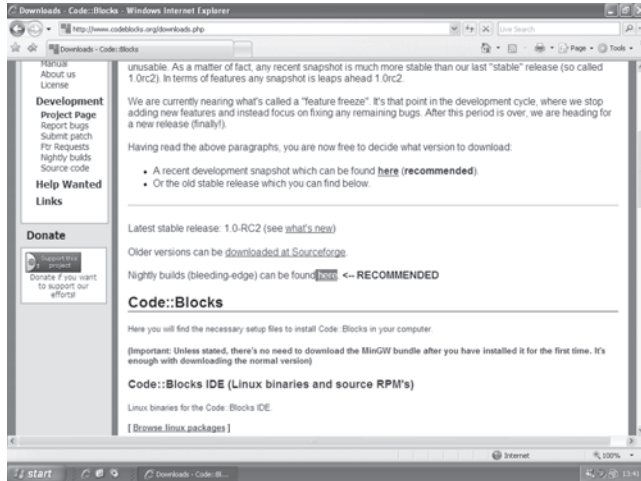


Figure 3-14

- For each Code::Blocks nightly build a total of three files (each in 7-Zip format) should be downloaded to the local computer, as follows:
 - **wxmsw28u_gcc_cb_wx284.7z** — 7-Zip archive featuring the Code::Blocks DLL dependency, wxWidgets interface library.
 - **mingwm10.7z** — 7-Zip archive featuring another DLL Code::Blocks dependency for working with MinGW compilers.
 - **CB_rev2_win32.7z file** — 7-Zip archive featuring the Code::Blocks C++ IDE, documentation, application, and other associated files.

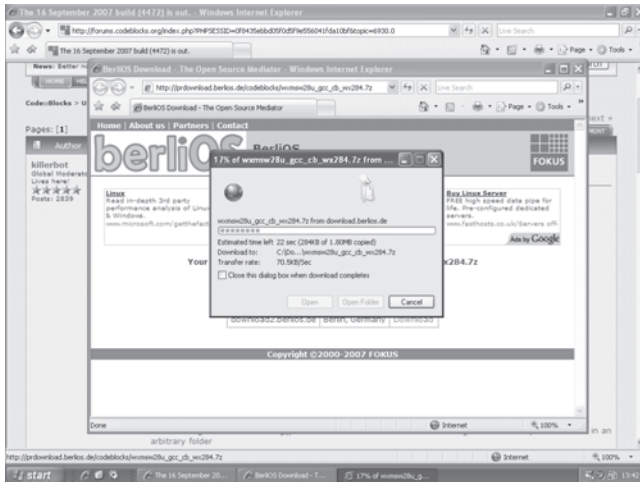


Figure 3-15: Download the Code::Blocks nightly build files.

- Using 7-Zip (as downloaded in step 1), extract all the contents of each 7-Zip archive into the same folder on the local computer (e.g., C:\CodeBlocks).

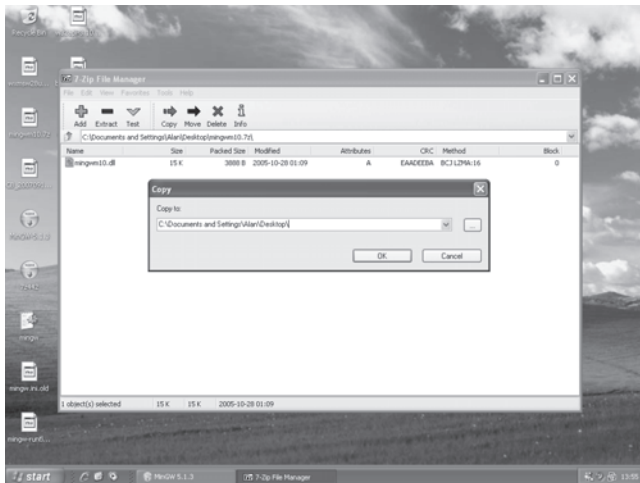


Figure 3-16: Extract the archive files.

- Code::Blocks is now installed and ready to run, and can be executed from the Code::Blocks folder whereupon the MinGW compiler should be detected by Code::Blocks successfully. For simplicity, users could further add a shortcut to the Code::Blocks executable in the Windows Start menu.

3.4.1 Code::Blocks Projects

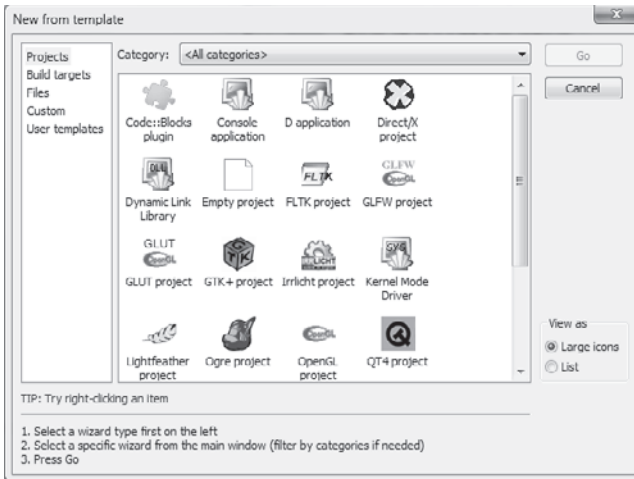


Figure 3-18:
Code::Blocks Wizard
options

Like a Visual Studio .NET project, a Code::Blocks project broadly refers to an organized collection of C++ source and header files and a series of compiler settings (such as which compiler to use), which together can compile into an executable application for any supported Code::Blocks platform. In short, one Code::Blocks project equals one application, though other configurations are also possible. Using Code::Blocks, projects may be created empty from scratch, where developers add source files and manually define compiler settings according to their requirements, or projects may be created as specified types from a series of premade templates using the Code::Blocks wizard. Here, each template automatically generates projects complete with the appropriate compiler settings and skeleton source code necessary for starting whatever kind of project the template is designed to build (such as a computer game project). Both blank and template-based projects are created using the Project Wizard, accessible from the Code::Blocks menu via File | New | Project. A selection of available project templates are listed below, especially those of relevance to game developers.

- **Empty project** — Typically used to create C++ projects for which there is no available Code::Blocks template, this wizard creates a blank, named project with no initial source or header files that adopts the default compiler settings.
- **SDL project** — SDL is an acronym for Simple DirectMedia Layer and is a free, open-source software development kit (or library) used for creating cross-platform games. The Code::Blocks SDL application wizard generates and configures Code::Blocks projects for use with SDL (assuming the SDL library is installed to the local machine already). Chapter 5 examines game creation with SDL in Code::Blocks.
- **OGRE project** — OGRE (Object-oriented Graphics Rendering Engine) is a free, open-source, and high-powered 3D software development kit designed primarily, though not exclusively, for making real-time 3D games complete with shaders and similar effects. Like the SDL wizard, the Code::Blocks OGRE wizard generates C++ projects configured and ready to run using the OGRE library, but assumes OGRE as a library (its source files, etc.) is installed to the local computer already. Chapter 10 examines game creation with OGRE in Code::Blocks.
- **OpenGL project** — First developed by Silicon Graphics, Inc. in 1992, OpenGL (Open Graphics Library) is one of the industry standard, cross-language, and cross-platform graphics rendering architectures for developing applications with real-time 3D graphics, including games, virtual reality software, and simulations. Code::Blocks offers an OpenGL project wizard for creating OpenGL games, though neither the OpenGL project wizard nor OpenGL is considered in more detail in this book.

3.5 Cross-Platform “Hello World” Application

The famous “Hello World” application that does nothing more than print to the screen what its title implies is customarily the first program a fledgling programmer creates. So following this tradition for no reason in particular, we’ll create a cross-platform “Hello World” application for Ubuntu and Windows using Code::Blocks. The more rebellious readers may prefer to break tradition and print something different! The Hello World project may be created and compiled initially in either the Windows or Ubuntu distribution of Code::Blocks, and then later migrated and compiled in the other, thereby creating two binary distributions of Hello World from the same source code, one for each platform. This book begins the Hello World project with Code::Blocks in Windows and subsequently migrates to Ubuntu, as illustrated in the following steps.

1. Beginning from the Windows desktop, start Code::Blocks and create a new **Console project** from the Application Wizard. Then choose the **Create a new project** link.

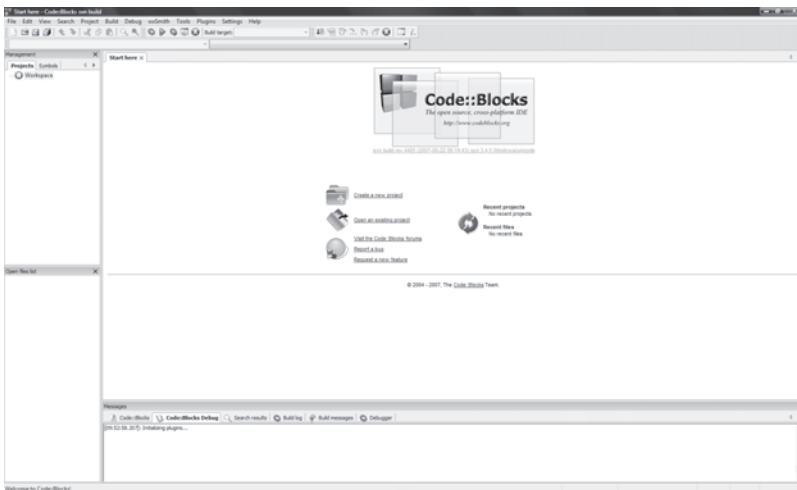


Figure 3-19:
Choose

2. Enter project details, including the project name and a valid local path in which to save the associated source files. Click **Next**.

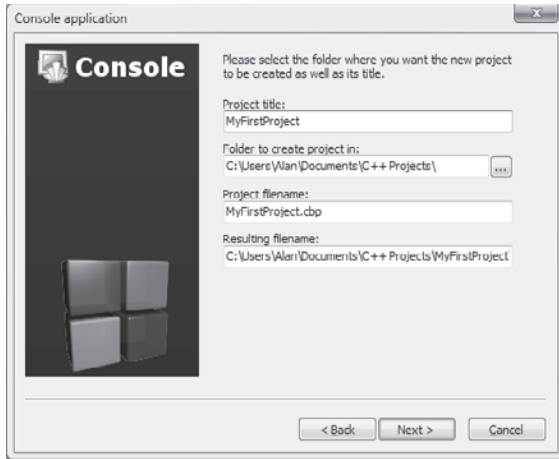


Figure 3-20

3. Accept the default compiler settings by clicking **Next**.

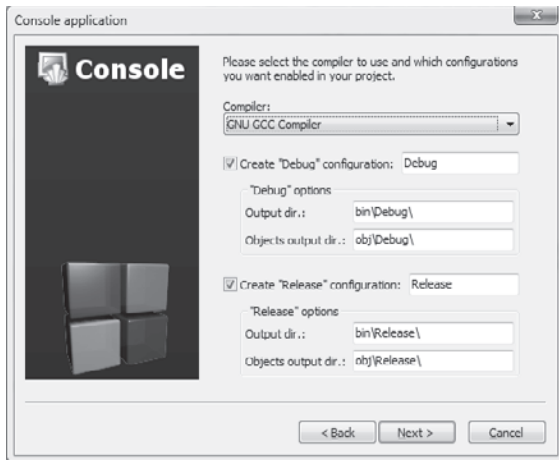


Figure 3-21

A Hello World project complete with source code is now generated, as follows:

```
#include <iostream>

using namespace std;
```

```
int main()
{
    cout << "Hello world!" << endl;
    return 0;
}
```

4. Select **Build | Run** from the Code::Blocks main menu, or click the **Build** icon from the Code::Blocks toolbar, to compile and execute the Hello World project.

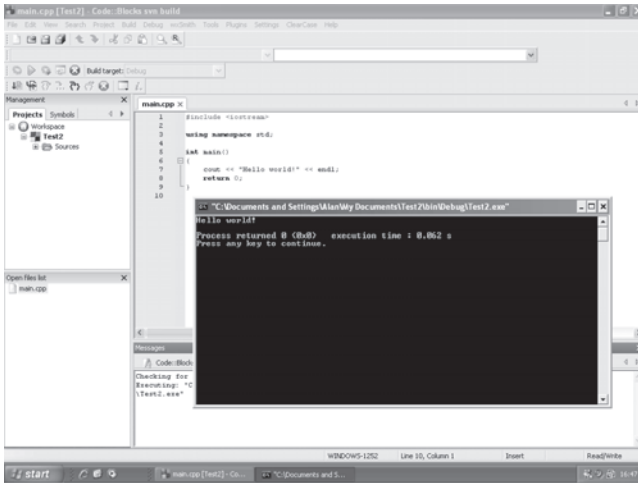


Figure 3-22: The Hello World! Project in Windows

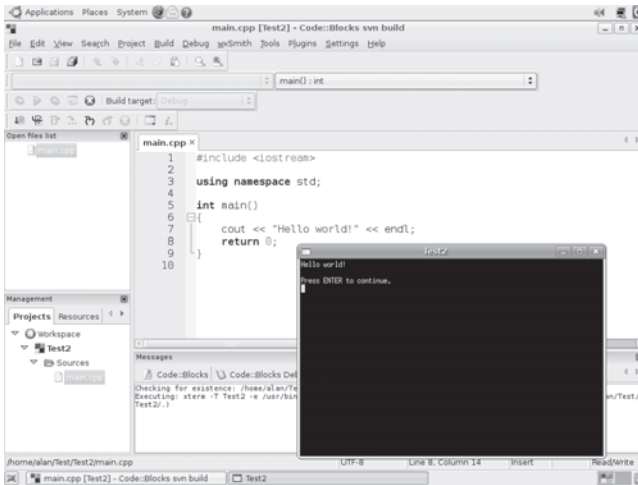


Figure 3-23: The Hello World! Project in Ubuntu

3.6 Graphics and GIMP

First created by Spencer Kimball and Peter Mattis in 1995 and then made available on UNIX/Linux-based systems, GIMP (GNU Image Manipulation Program) is often considered a free, cross-platform, and open-source equivalent to the popular Adobe Photoshop, an application for editing, retouching, and reworking photographs or 2D images (bitmaps, .pngs, .jpegs, etc.). GIMP is cross-platform insofar as it supports Windows, Linux, Mac, FreeBSD, and Solaris. GIMP is also fast becoming one of the most popular and widely used image editing tools, becoming the standard (default) image editor for a variety of Linux distributions including Ubuntu, Mandriva, SUSE, and Fedora. GIMP is useful for cross-platform game development because developers can use it to perform at least the following tasks:

- Retouch textures and other in-game images using the GIMP brushes, inks, stamps, and other editing tools. It can be used on photos as well as 2D renders from 3D software like Blender 3D (discussed in detail in Section 3.7).
- In collaboration with 3D rendering software or real-time 3D games, GIMP is often used by developers to create “seamless” tileable textures that can be texture mapped onto 3D objects. A texture that is *seamless* is one whose opposite edges (left and right, and top and bottom) are identical to the other so that the texture may be repeated (or tiled, or juxtaposed) across a 3D surface (like a cube face or a wall) without lines or breakages appearing where the edges meet between any two tiles or any two repetitions. A *stochastic* texture is one whose pixels are a random combination of brightness, contrast, and color.

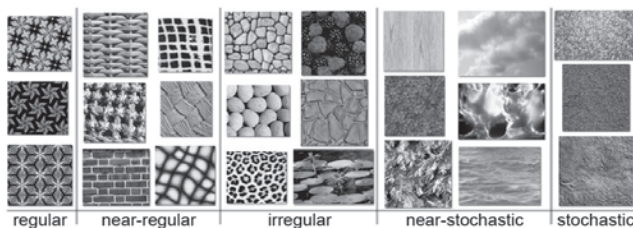


Figure 3-24: A selection of tileable textures

- Define image alpha channels and transparency. Images that feature transparent or semi-transparent regions often use alpha channels. An *alpha channel* is a separate canvas (or image) hidden and embedded inside an image that is equal in pixel width and height to the visible, standard layer of the image. The alpha channel's pixels may only be grayscale, ranging from black to white or any shade between. The pixels of the alpha channel map 1:1 to the pixels in the main image, and the color of each pixel determines the transparency of the corresponding pixel in the main image with black = transparent, white = opaque, gray = 50% transparent, etc.

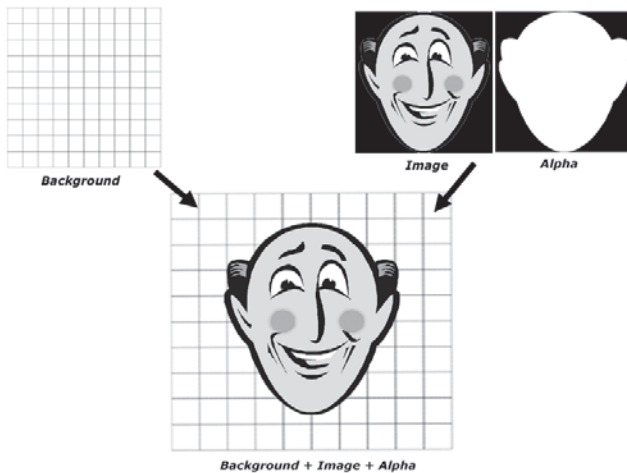


Figure 3-25: An image with an alpha channel

3.6.1 Installing GIMP on Windows or Mac

GIMP is preinstalled with Ubuntu and is available from the main menu via Applications | Accessories | Graphics | GIMP. This section explains how to install GIMP on the Windows or Mac platform.

1. Beginning from the desktop, navigate a web browser to the GIMP home page.
 - For Windows: <http://www.gimp.org/>
 - For Mac: <http://wilber-loves-apple.org/>

2. From the home page, download the GIMP package to the local computer.
3. Once downloaded, run the GIMP installer.

3.6.2 Using GIMP

This section offers an overview of GIMP and demonstrates how to perform some common tasks encountered in game development. More detailed tutorials and guides on GIMP can be found at the following web addresses:

- <http://docs.gimp.org/>
- <http://wiki.gimp.org/gimp/>
- <http://www.gimp.org/tutorials/>
- <http://www.gimp-tutorials.com/>

3.6.2.1 Creating Tileable Textures Using GIMP

Tileable textures are those images that may be repeated seamlessly, like bathroom or kitchen tiles, arranged one beside another in columns and rows across a plane surface. For any tile to repeat “seamlessly” (without a visible edge where any two tiles meet side by side), the pixels on the edges of a tile must be identical or “connectable” to the pixels on its opposite edge (left and right, top and bottom). The following step-by-step procedure explains in detail how to create a seamless image using GIMP.

1. Start the GIMP application and create a new, blank image 300 pixels in width and 300 pixels in height.

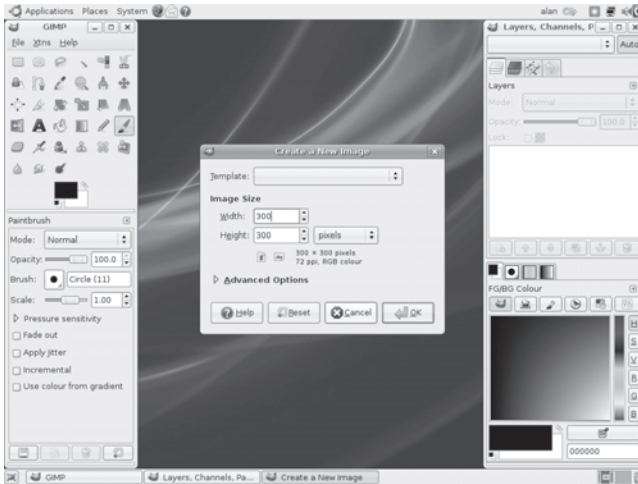


Figure 3-26: Create a new image.

2. Fill the image with random pixels, using the brush, fill, or pen tools. The sample image in Figure 3-27 is used for demonstration purposes only; most developers will work with meaningful images loaded from files on disk.

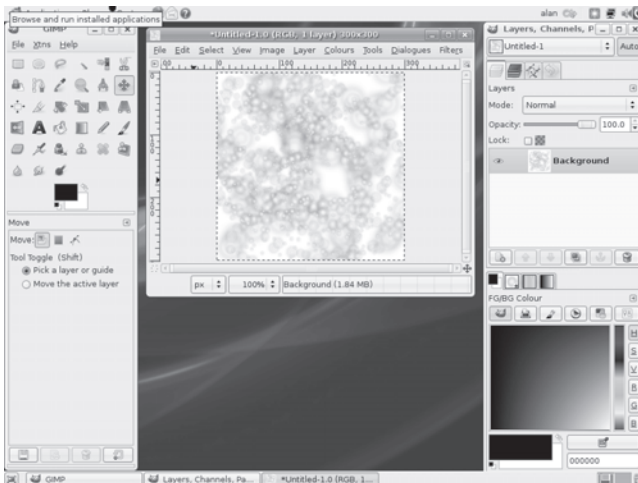


Figure 3-27: Fill the image with random pixels.

3. Make the image seamless by choosing **Filters | Map | Make Seamless** from the main menu.

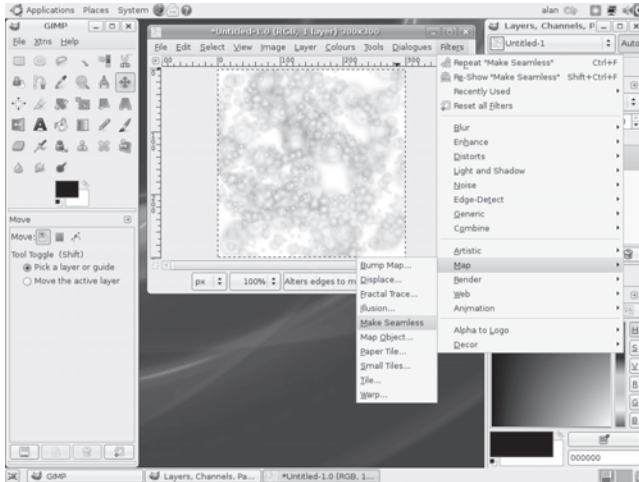


Figure 3-28: Choose **Filters | Map | Make Seamless**.

4. Test the seamless tile by creating a new image that is several times the width and height of the original, and then copying and pasting the smaller original three times into the larger, arranging the tiles in a grid to see the seamless tile effect.

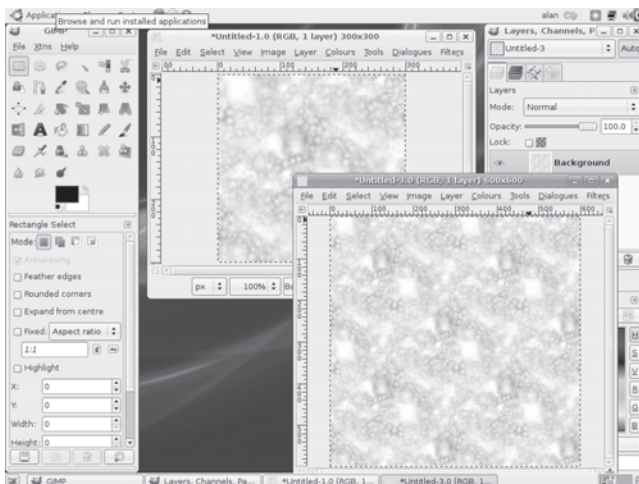


Figure 3-29: Copy the original image several times and arrange the copies side by side in a grid.

3.6.2.2 Editing Image Transparency Using GIMP

This section highlights how to use GIMP's image alpha channels (or layer masks) to make regions of a GIMP image transparent or semi-transparent.

1. Start GIMP and create a new image 600 pixels in width and 600 pixels in height by selecting **File | New** from the main menu.

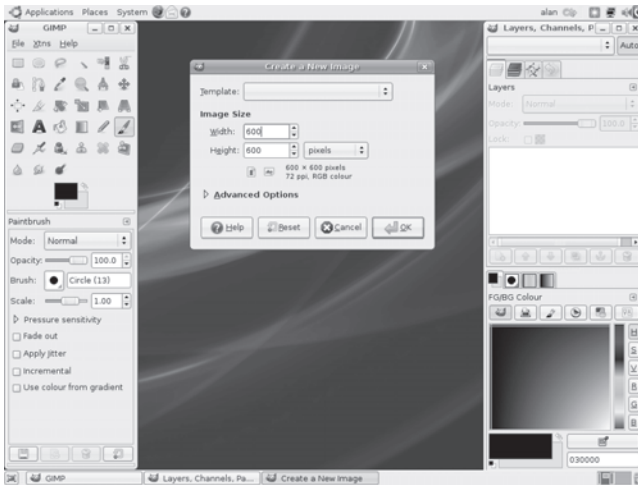


Figure 3-30: Create a new image that is 600 x 600 pixels.

2. Import an image or draw a design or pattern with regions that will later become transparent (such as is seen in Figure 3-31).



Figure 3-31

3. To add an alpha channel to the current image, select **Layer | Mask | Add Layer Mask** from the main menu.

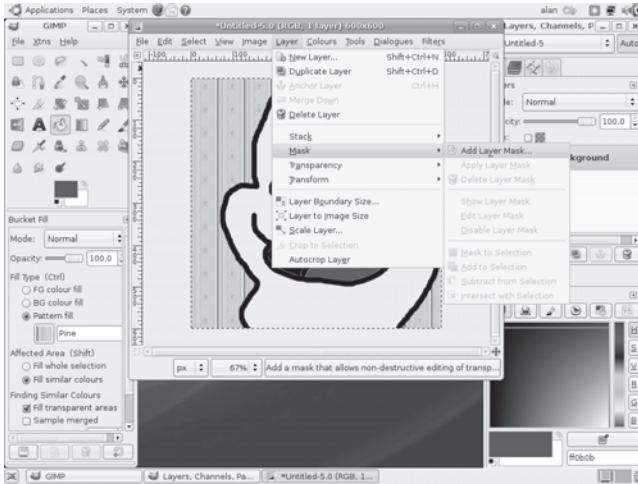


Figure 3-32: Choose **Add Layer Mask**.

4. From the Add Layer Mask dialog, choose **White (full opacity)** to make the image layer fully opaque.

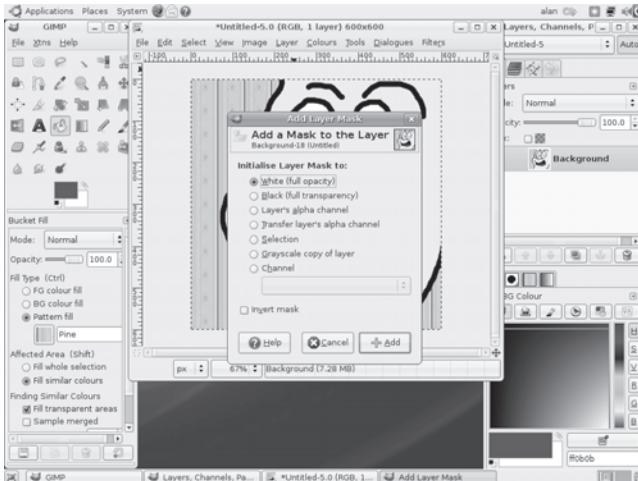


Figure 3-33: Choose **White (full opacity)**.

5. Select the new layer mask by clicking the white thumbnail in the Layers palette.



Figure 3-34: Select the white layer mask.

6. Afterward, use the standard GIMP paint tools to paint the transparency data into the image. Remember that the black pixels will be transparent, the white pixels will be opaque, and 50% gray pixels will be 50% transparent.



Figure 3-35: Start painting areas that you want transparent.

7. Finally, save the image in an image format that supports transparency, such as .bmp, .tga or .png.

3.7 Blender 3D

Like GIMP, Blender 3D is free, cross-platform, and open-source. Blender 3D is often compared favorably to commercial 3D rendering and animation software such as Autodesk's 3ds Max, LightWave, and Maya, each of which are popular choices among game developers. Supporting a host of different platforms, including Windows, Mac, Linux, FreeBSD, Solaris, and IRIX, Blender 3D is a comprehensive 3D rendering suite designed for creating prerendered 3D stills and animations. The following sections show how to install Blender for both Windows/Mac and Linux Ubuntu. The intricacies of using Blender 3D are beyond the scope of this book, which focuses mainly on programming rather than graphics, but more information regarding Blender 3D can be found at the following web sites:

- Blender 3D e-book: http://en.wikibooks.org/wiki/Blender_3D:_Noob_to_Pro
- Blender tutorials: <http://www.blender.org/education-help/tutorials/>
- Blender video tutorials: <http://www.blender.org/education-help/video-tutorials/>
- Blender 3D manual: <http://wiki.blender.org/index.php/Manual>

3.7.1 Installing Blender 3D on Linux Ubuntu

1. Beginning from the Ubuntu desktop, launch the Synaptic Package Manager from the Ubuntu main menu by selecting **System | Administration | Synaptic Package Manager**.
 2. Search for “Blender” and mark the Blender items listed in the Ubuntu application repository. Then click **Apply** to install Blender 3D to the system.
-

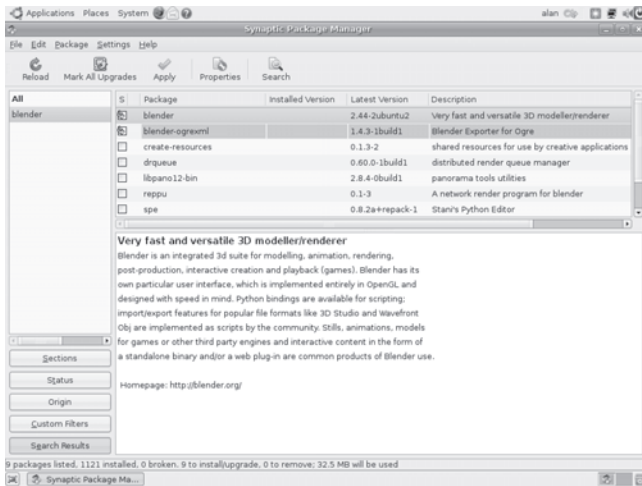


Figure 3-36

- Once installed, Blender can be launched from the Ubuntu main menu by selecting **Applications | Graphics | Blender 3D**.

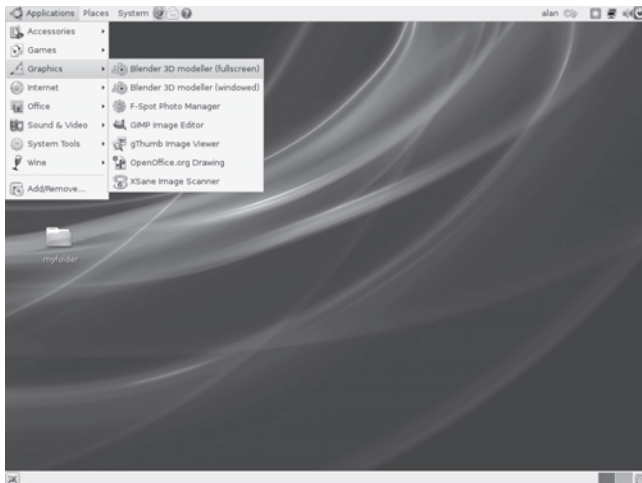


Figure 3-37

Blender 3D is now installed and ready to use.

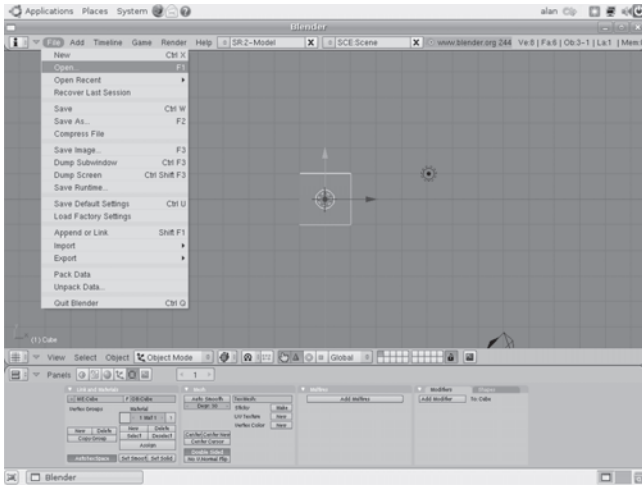


Figure 3-38

3.7.2 Installing Blender 3D on Windows/Mac

1. Beginning from the desktop, navigate a web browser to the Blender home page at <http://www.blender.org/>.

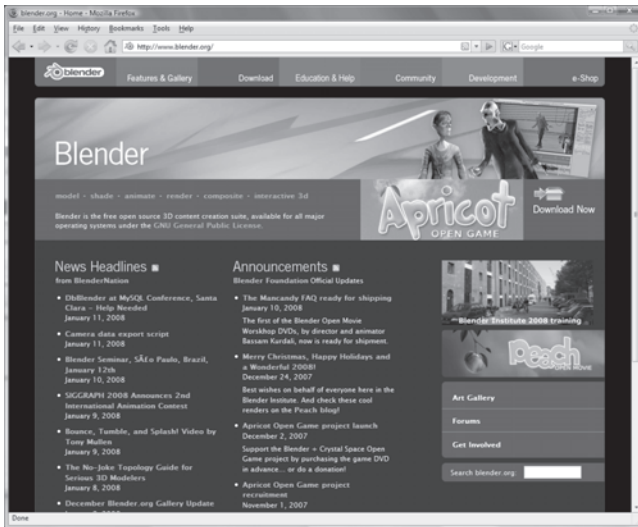


Figure 3-39

- At the Blender site, click the **Download** button from the top menu bar.

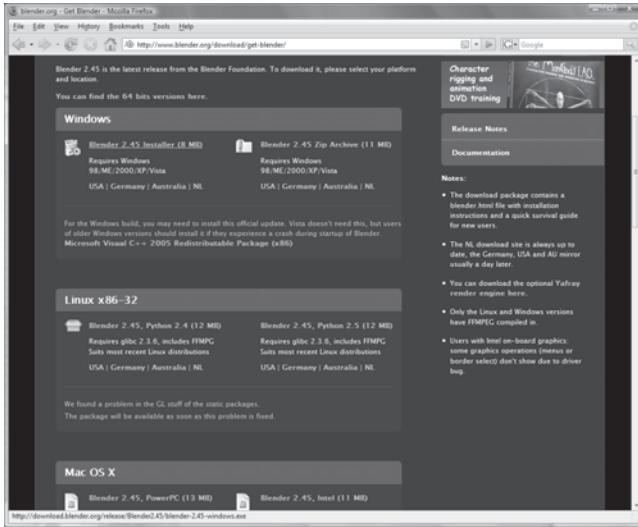


Figure 3-40

- Once at the download page, choose the appropriate Blender distribution for your operating system, and download the package from the web to the local machine.
- Once downloaded, run the installer to install Blender to the system.



Figure 3-41

Blender 3D is now installed and ready to run.

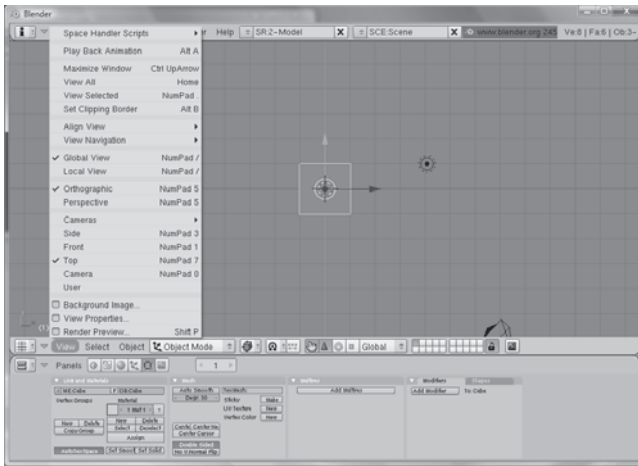


Figure 3-42

3.8 Conclusion

In summary, this chapter has detailed how to use three free, open-source, and cross-platform game development tools: the Code::Blocks C++ IDE, the GIMP image editor, and Blender 3D. GIMP and Blender 3D are products important for game graphics, but the main focus of this book is game development in terms of programming, and as such these two graphics applications are not considered in further detail in this book. Code::Blocks, however, is considered further in later chapters.

In the following chapter we'll discuss the basics of cross-platform game development.

This page intentionally left blank.

Chapter 4

Game Programming Basics

By now readers will probably have configured a cross-platform development software and hardware setup for making and debugging their cross-platform games. Chapter 1 illustrated at least three methods for configuring a cross-platform setup: running several machines, each with a different OS on a *different* machine; running multiple OSs on a single machine through dual-booting; and running multiple guest OSs on a single host OS through virtualization. In addition to this cross-platform infrastructure (whichever is chosen), the open-source and freely available C++ IDE, Code::Blocks, was selected as the primary development suite for compiling cross-platform games. Having then selected this IDE, and so thereby having selected C++ as the primary development language, this chapter considers the next steps. Specifically, this chapter considers in detail the following key developmental issues:

- The basics of game programming and how to get started at making a game. We need to know where to begin, features that are common to all games on all platforms, common game algorithms, and data structures that are useful for games.
- Selecting suitable game libraries. Beyond C++ as a language itself, developers make use of third-party game libraries that “plug into” the language and make their programming lives easier. These libraries are collections of classes and functions made by other programmers for use by programmers, and they are designed to meet specific developmental needs. Accordingly, game developers make use of graphics libraries for rendering pixels and

images to the window, sound libraries for playing sound and music to the speakers, physics libraries to simulate physical reactions, and others. This chapter considers a selection of different libraries available to developers and determines each library's suitability for developing cross-platform games, based primarily on the features they offer and the platforms they support.

4.1 Game Programming — Getting Started

Before a developer actually sits down to code a game based on his latest fantastic idea, he will have previously brainstormed a “design” on paper in the form of a game design document, which at the very least helps to organize thoughts. The question is how detailed should this document be. It is possible (indeed, often happens) to overplan and to spend so much time at the drawing board plotting and planning for all minutiae and every eventuality that it becomes both easy and dangerous to lose oneself in a mountain of plans, a mountain from which it becomes costly to escape in terms of both time and money. But this of course can be no less dangerous to a project than underplanning; the solution is therefore a case of getting the balance right between things that are planned and things that are allowed to happen unplanned, with each being as important as the other since games rarely go 100% according to plan. Of course, developers will not be making the game up spontaneously as they code, like a person writing a letter, but will instead aim to follow a carefully planned road map, setting out in advance a directed and concise strategy defining, among other details, the objective of the game, the enemies, the maps, the characters, the time frame, the budget, and even the platforms to be supported by the final product. Let's consider these details further.

4.1.1 Genre and Objective

Each game is said to belong to at least one *genre* insofar as the content of any game may be described as being puzzle, action, adventure, sport, first-person shooter, etc. Similarly, the *objective* of a game refers to its purpose, the ultimate aim to which a gamer is playing to meet, such as “in every level a gamer must collect a piece of an ancient artifact such that by the final level all the pieces may together be reassembled.” To frame the context for development then, the game design document (GDD) should state clearly the genre (the market niche) of the game as well as the objective (purpose) of the game from the perspective of the gamer.

4.1.2 Time Frame and Budget

In terms of commercial game development, the expression “time is money” highlights the intimate relationship between time (the timeline of development) on the one hand, and the cost (cost of time and resources) on the other. Specifically, work like game development requires “resources” such as software and human effort as well as “time” such as time-to-completion; so work then represents an investment of money. Thus, game design documents should first and foremost specify a budget — that is, the maximum sum of money allocated to the development of a single game. Based on this budget, the cost of time and resources can be figured into a complete timeline or workflow diagram that charts the development of a game. To give some typical (and only approximate!) examples: It is not unusual for a team of 30 people working five-day weeks to require 18 months to develop a full, big-budget AAA game from start to finish. With a smaller budget and an independent team of two to seven people working part-time, it requires 18 to 24 months to make a medium-sized puzzle adventure (e.g., Teenage Lawnmower, Dr. Lunatic Supreme with Cheese, Zombie Smashers, etc.).

4.1.3 Game Ideas

Developers looking for ideas and inspiration concerning the genre to which their planned game should follow might like to consider some of the following most common game genres:

- **RPG (Role-Playing Game)** — Daimonin, Oblivion, Final Fantasy, and Might & Magic are just some among a broad field of games known as RPGs, or sometimes referred to as CRPGs (computer RPGs). In these games, gamers create and control a character or party of characters who live and act within an immersing and fictional world, typically featuring wizards, goblins, dragons, and other fantasy creatures. The abilities and success of each character in the world is determined largely by its qualities and skills — such as Strength, Intelligence, and Health — and these qualities are presented to the player statistically as factors on a character sheet. Throughout the game, a character’s success in meeting the challenges of the game world (completing quests, etc.) is typically rewarded by receiving points. Subsequently, the player may carefully distribute (spend) the points across a character’s skill set, weighting each skill so as to improve the character as whole. As a result, RPGs are driven more by character development through the accumulation of points than by any directed, linear progression from a beginning to an end; and consequently, many RPGs are said to be “open-ended” insofar as they cannot be “completed.”

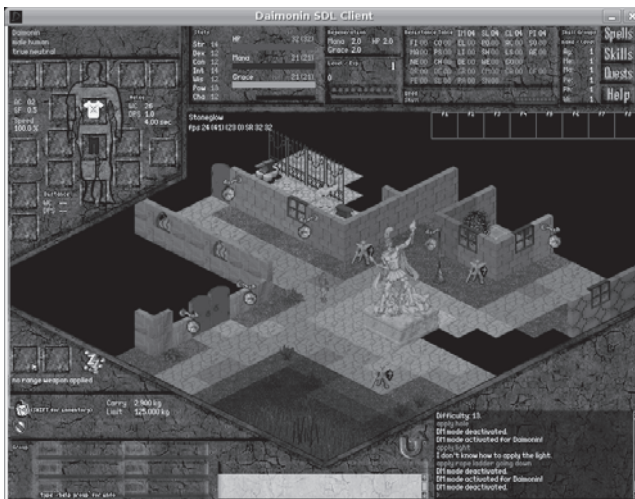


Figure 4-1: Screenshot from Daimonin, a free, cross-platform, and open-source RPG, available at <http://www.daimonin.net/>

- **FPS (First-Person Shooter)** — Doom, Quake, Unreal Tournament, Half-Life, and Halo are but a few of the most popular FPS games, and games from this genre are some of the best selling in the industry. The online encyclopedia Wikipedia perhaps defines an FPS best as “a video game that renders the game world from the visual perspective of the player character and tests the player’s skill in aiming guns or other projectile weapons [at enemy targets].” In short, most FPS games center around shooting people or monsters. It is due to their characteristic violence that many have stirred up controversy, arousing the criticism of both anti-game lobbyists and psychologists.

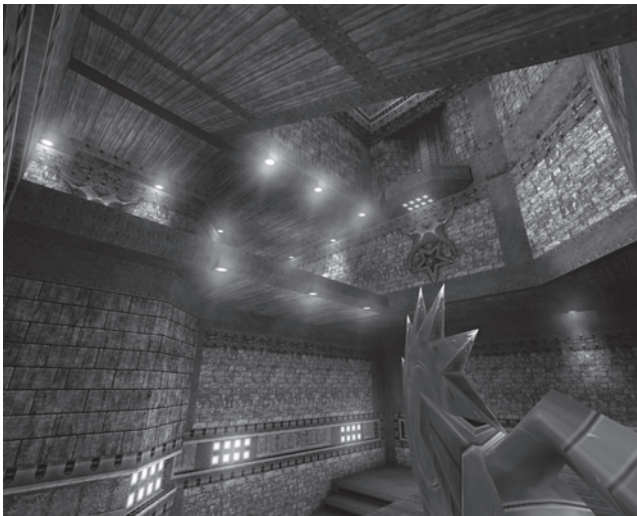


Figure 4-2: Screenshot from OpenArena, a free, cross-platform, and open-source first-person shooter, available at <http://www.openarena.ws/>

- **Platformers** — The term “platformer” (or platform game) describes an action-packed genre, so called because most platform games feature a player-controlled character that must navigate carefully through a hostile environment from the beginning to an end point without coming to harm. This often involves running and jumping from location to location while combatting enemies, avoiding traps, and collecting power-ups. According to standard video game practice (and particularly so in platformers where the player is at risk of pitfalls and dangers), the player is often equipped with three or more “lives,” with each life representing one opportunity at successfully completing a level without failing (that is, without

dying). A death, then, corresponds to the loss of one life, and the loss of all lives through successive failures results in “game over.” In recent years, however, most developers have all but discarded the traditional “three lives” structure for their games in favor of an “infinite attempts” model, hoping to make them more accessible to a wider variety of gamers. Some of the old-school platformers include Super Mario Bros., Sonic the Hedgehog, Zool, Fire and Ice, Earthworm Jim, and Jazz Jackrabbit. Newer platformers include Psychonauts, Supercow, and Crash Bandicoot.

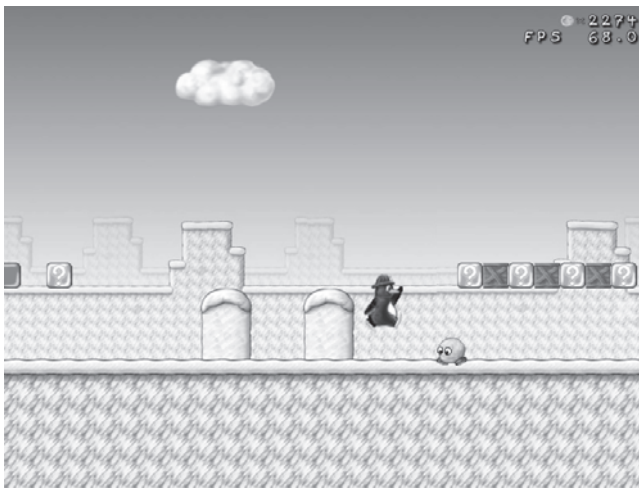


Figure 4-3: Screenshot from SuperTux, a free software platform game, available at <http://supertux.lethargik.org/>

- **RTS (Real-Time Strategy) and TBS (Turn-Based Strategy)** — Military logistics and strategic planning are at the heart of the strategy genre. RTS games include the likes of Command & Conquer, StarCraft II, and Darwinia, while TBS games include a list of titles such as Heroes of Might and Magic, The Battle for Wesnoth, and UFO: Alien Invasion. Both the real-time and turn-based variants of the strategy genre commonly put the player in military command of a faction (or army) pitted against other AI factions in a game world where each faction strives for complete domination over a common territory and common resources. In the turn-based variant of the strategy game, the battle between factions takes place across a staggered series of turns, where the commander of each faction (player or AI) delivers orders to their units and responds to enemy actions only during their allocated turn cycle.
-

This results in cool and calculated game play; that is, before beginning every new turn the player must first wait for each other faction to complete their corresponding turn, and the enemy factions in turn wait for the player and others to complete theirs. In the real-time variant, by contrast, each faction plans and acts simultaneously in action-packed real time, requiring both quick reflexes and quick thinking.



Figure 4-4: Screenshot from Glest, a free and cross-platform RTS game, available at <http://www.glest.org/>

- **Casual Games** — Statistically, casual games represent a growing market in which both independent developers and lower-budget teams (largely from Europe, Latin America, and Japan) have flourished since the spread of broadband and faster domestic Internet access. The title “casual game” is an umbrella term referring to a broad variety of games. These range from games like Tetris to Diner Dash, many of which are sold and distributed by download to gamers via online gaming portals such as Reflexive Arcade and Big Fish Games. Though any single casual game may have almost no feature in common with any other casual game on the market, most are united insofar as they are designed to be “lightweight,” designed for mass appeal, and intended to be original. A game is generally said to be lightweight if it is quick for gamers to download from the web (small in MB size) and if it requires few system resources to execute successfully on the gamer’s computer. A game has mass appeal when it comes with an attractive price tag

and markets itself demographically to the largest gamer population (men, women, young, old), instead of to a niche market like first-person shooters.



Figure 4-5: Screenshot from the casual game Pingus, inspired by Lemmings; free and cross-platform, and available at <http://pingus.seul.org/>

4.2 Preparing to Make Games

Before firing up Code::Blocks to code a game according to the design document, any developer who first sits back to carefully consider the basics, or fundamental building blocks, of any games that spring to mind will likely notice a number of common features between them. Some of these features are worth considering further, even though their existence and implementation may appear trivial or obvious at first sight.

- First, every game in any genre (from FPSs to RPGs) must work with data; that is, strings, integers, floats, etc. For example, a game may display text at an X,Y coordinate on the screen, calculate differences between player scores, read player input from the keyboard, read both numerical and textual data from saved game files in order to restore sessions saved by the gamer on previous occasions, and handle many other situations. In common with most software, therefore, a game is at its most basic level a data-handling machine. It accepts incoming data, processes it, and outputs
-

a result. Consequently, developers often find it useful to standardize the data types used throughout their source code for any given game. For example, they may use only one data type for all strings, such as the cross-platform STL class `std::string`, instead of using a variety of string data types for different string variables like `char`, `CString`, `std::string`, etc. Class `std::string` and the STL more generally are considered in more detail in Section 4.3.

- Second, since all games collect and process data at run time, whether from files on disk, user input, or the result of additional calculations (as described in the previous chapter), it is important for developers to show foresight when making their games by designing algorithms to structure and process data optimally in memory. For example, in a typical RTS game like StarCraft or Command & Conquer, each faction (including the gamer's) must eliminate all rival factions for domination of a given territory. In developing their army, each faction begins by first harvesting nearby resources (such as wood, ore, and gold) in order to fuel the construction of buildings and technology, and from these to ultimately recruit more units, and to recruit and deploy more creatures and fighters, who are subsequently dispatched across the map to eliminate enemy targets. Here, then, each faction in the course of its in-game development collects at least three kinds of items: resources, buildings, and units. A developer is then faced with the problem of how best to code (create) three separate lists in memory, each designed to keep track of a faction's resources, buildings, and units at run time. A developer may initially choose, for example, to create a fixed-size array for each faction to hold a collection of pointers to its units, where each element in the array is a pointer to a single military unit (a wizard, a goblin, a tank, etc.). However, a problem arises for the developer when he considers that as new units are created by the gamer at run time, those units must be added into the last vacant elements of the units array, and as units are destroyed by, say, enemy fire, they should then be removed from the array without affecting any other active elements. In short, the developer requires an array class that may dynamically grow or shrink in memory as items are added and removed, meaning that the array is always sized exactly to hold no more or less than the number of items in memory at

any one time. This kind of list arrangement can be achieved using the STL class `std::vector`, which is discussed in a later section of this chapter.

- Next, most games are driven by a “message pump” (or life line) that is qualitatively different from the event-driven programming used in other non-game software, and consequently it has earned the name “game loop.” Almost all non-game software (such as word processors, database applications, or graphics editors) work by listening for and responding to user input (such as keypresses and mouse movements) as they occur at run time when they are sent to the application window through standard Win API messages. For example, a word processor only prints a document when the user clicks the Print button, and it only inserts characters into the active document when the user presses keys on the keyboard; if the user does nothing, then the program does nothing except wait for input from the user. Games, however, work differently, which can be illustrated by an example. In a typical first-person shooter game such as Quake, Doom, or Unreal, the player is armed with a weapon and thrown into an arena with other competitors who each must deploy their aiming skills and stealthy tactics against one another to become the last man standing. Here, like non-gaming software, the player character shoots when the user presses the Fire button, and jumps when the user presses the Jump button. But, unlike event-driven software, the enemy combatants and other game events are occurring *simultaneously* with all other events such that, if the player stood still and the user did nothing, the game wouldn’t stop and the opponents wouldn’t freeze waiting for the user to press a key. Instead, the game continues as normal, and the NPCs (non-player characters) continue to participate as though they were “real” humans, whether the player is participating or not (unless the user presses a Pause button). For this reason, games are usually driven by a “message loop” mechanism rather than by an event-driven framework since game action occurs in real time and in no way depends on the user’s input to continue working. The game loop is considered in more detail in a later section of this chapter.
-

4.3 Using the STL: Strings and Lists

To summarize the preceding section on game development basics: Game development for a programmer begins from a library, or a common framework of data structures, algorithms, and functions. And for cross-platform C++ games specifically, the STL (Standard Template Library) offers just such a comprehensive set of tools, particularly in the form of classes such as `std::string` (for strings) and `std::vector` (for lists of objects and lists of pointers to objects). The importance of these two classes for game development is now considered more closely.

4.3.1 `std::string`

A *string* is a linear array of characters arranged sequentially in memory; for example, the word “hello” is a string of characters, where each character is a letter in the word. Typically, C++ strings are declared literally as an array of chars (e.g., `char mystring[50]`), but the STL string class makes this process simpler. The following step-by-step tutorial for using class `std::string` shows how to work with C++ strings in a new way and is designed to be both a gentle introduction to programming with `Code::Blocks` and C++, and a user-friendly guide to make working with strings easy. We’ll be creating strings, copying strings, manipulating strings, and more.



NOTE. An exercise in processing strings and characters of strings (like the samples that follow) may prove helpful for any developer who may later choose to use XML for storing data on disk, external to the game. XML files are essentially organized text files, and so XML strings — like those used for XML properties and tags in the file — may require processing like any other string. This means `std::string` as a class will likely be important for things such as processing saved game files and data files.

4.3.1.1 Configuring Projects to Use STL and `std::string` with `Code::Blocks`

1. Beginning from the desktop, start the `Code::Blocks` IDE and use the New Project Wizard to create a new console (shell/command line) project ready to compile. See Chapter 3 for more details on using `Code::Blocks` if needed.
2. Open the main project source file (.cpp) and add the string header shown in bold to the end of the existing preprocessor directives. This directive includes the STL `std::string` class header so `std::string` may be used throughout the project.

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
    cout << "Hello world!" << endl;
    return 0;
}
```

3. Save the `Code::Blocks` project by choosing **File | Save**; then compile the active project by clicking the gear icon on the toolbar or by choosing **Build | Compile**.

The project is now configured to use class `std::string`.

4.3.1.2 Declaring, Creating, and Assigning Strings with `std::string`

Instances of `std::string` can be created as follows:

```
//Empty
std::string MyString1;

//Set to hello
std::string MyString2 = "hello";

//Set to hello
std::string MyString3 = MyString2;
```

Any two or more instances of `std::string` may be concatenated (combined) together to form a single larger string using the standard addition (+) operator:

```
std::string MyString1 = "hello ";
std::string MyString2 = "world";

//MyString3 = "hello world"
std::string MyString3 = MyString1 + MyString2;
```

The value (contents) of any instance of `std::string` may be queried or determined by using the C++ equality operator (`==`):

```
std::string MyString1 = "hello world";

if(MyString1=="hello world")
{
    //Do something here
}
```

4.3.1.3 Looping through Characters of a String with `std::string`

Each character (letter) in a single instance of `std::string` can be read individually using the standard array subscript operator (`[]`) along with an array index specifying the character offset into the string, counting from left to right from the first character at (0) as follows:

```
std::string MyString1 = "hello world";

cout<<MyString1[3];
```

The `length` method of `std::string` returns the length of the string (i.e., the total number of characters from which the string is composed). This method, in combination with access to individual characters in the string using the array subscript operator, means any `std::string` can be iterated through character by character.

```
std::string MyString1 = "hello world";

for(int i = 0; i < mystring1.length(); i++)
{
    cout<<mystring1[i] << "\n";
}
```

Each character in a string may also be iterated through by using STL iterators rather than array indexes, where the `begin` method returns a pointer to the first character, and the `end` method returns a pointer to the last character, as follows:

```
std::string MyString1 = "hello world";
string::iterator my_iter;
for(my_iter = MyString1.begin(); my_iter != MyString1.end();
my_iter++)
{
    cout<<*my_iter;
}
```

4.3.1.4 Searching for Characters in a Specified Instance of `std::string`

For situations in which the content of a string is unknown, or where a string's structure and format must be analyzed closely, the `std::string` class offers the `find` method to search through a given string for a specified character or sequence of characters, returning a pointer to the first character in the string where a match is found. This method may be called as follows:

```
/*Counts all occurrences of the world hello in a specified
instance of std::string*/

string input;
int word_count = 0;

cout<<"Please enter a string now:>";
getline(cin, input, '\n');

for(int i = input.find("hello", 0); i != string::npos; i =
input.find("hello", i))
{
    word_count++;
    i++;
}
cout<<word_count;
```

4.3.1.5 Extracting and Inserting Substrings from and to a Specified Instance of `std::string`

A string is an array of characters, and a substring is a smaller subset of characters from that array; “ello”, “lo”, and “o” are all substrings of “hello”. To extract a substring (dest) from a specified larger string (source), the `std::string` class offers the `substr` method. This method returns a new string that is the requested substring, and it also accepts two integer arguments: one specifying the offset into the source string marking the first character of the substring (dest) and the other specifying the length of the substring in characters as measured from left to right from the offset. An example follows:

```
std::string MyString1 = "hello world";
//string is "o wo"
std::string substr = MyString1.substr(3, 7);
```

In addition to substring extraction using the `substr` method, a string of any length can also be inserted into any instance of `std::string` using the `insert` method. Similarly, strings can be removed from any instance of `std::string` using the `erase` method. The following code sample illustrates both insert and erase at work:

```
std::string MyString1 = "hello world";
//Now is "helthis is a substringlo world"
MyString1.insert(3, "this is a substring");
//Now is "hhis is a substringlo world"
MyString1.erase(1, 3);
```

4.3.1.6 Converting Instances of `std::string` to Standard `char*` Pointers

Despite the variety of benefits afforded by `std::string`, from features like substring extraction to character insertion, there will undoubtedly be moments when a developer encounters a function from a third-party library (such as a Win API call) that requires a string argument of type `char*`, and not of type `std::string`. Thus, so that instances of `std::string` may be type compatible with functions requiring arguments of type `char*`, the `c_str` method is offered to convert strings from type `std::string` to type `char*`. An example follows:


```
std::string MyString1 = "hello world";  
const char* MyString2 = MyString1.c_str();
```

4.3.2 **std::vector**

Most computer games keep track of lists of items. For example, RTS games (where factions fight one another for domination of a map) maintain at least three lists “under the hood” for each faction that participates in battle: one list for a faction’s resources (wood, ore, gold, etc.), one for its buildings (refinery, barracks, etc.), and one for its units (wizard, fighter, goblin, etc.). Similarly, in an adventure game like Monkey Island, Grim Fandango, or Syberia, gamers control a character that solves a mystery by collecting and using objects found around the game world. The objects collected by the player are added to their inventory (pockets) where they remain until they can be used or disposed of to further their progress in the game; and here, again, the inventory reveals itself to be a list of collected items in the same way a string is a list of collected characters.

The primary characteristics of a list are: Items can be added to or removed from the list at run time; and the list changes size in memory as items are added or removed in order to accommodate exactly the number of items it currently holds, no more or less (it is said to be *dynamic*). As it meets this criteria, the STL `std::vector` class offers game developers a template class for maintaining a dynamic list of items (of any data type) in memory. In short, `std::vector` is a class for holding a list in memory to which items can be added or removed at run time. Let’s examine how this class is used.

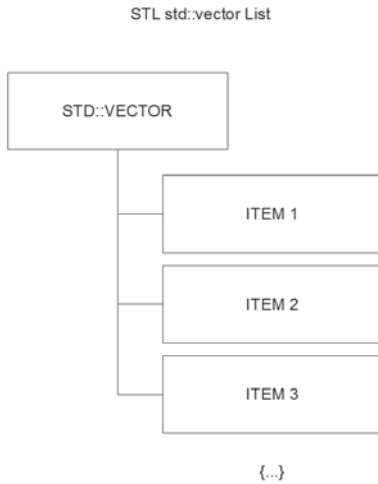


Figure 4-6

4.3.2.1 Creating a List with `std::vector`

1. Beginning from the desktop, start the Code::Blocks IDE and use the New Project Wizard to create a new console (shell/command line) project ready to compile. (See Chapter 3 for more details on using Code::Blocks.)
2. Open the main project source file (.cpp) and add the vector header shown in bold below to the end of the existing preprocessor directives. This directive includes the STL `std::vector` class header so `std::vector` may be used throughout the project wherever lists are required.

```

#include <iostream>
#include <vector>

using namespace std;

int main()
{
    cout << "Hello world!" << endl;
    return 0;
}
  
```

4.3.2.2 Declaring Instances of `std::vector`

The STL class `std::vector` is a template class, and each instance represents a unique list of objects (of any one data type) in memory. What that means is any single instance of `std::vector` is a list of objects of the same type: a list of integers, a list of strings, a list of pointers, etc. `std::vector` is said to be a template class because each instance (each list) must be declared as belonging to a specific data type at the time of declaration. Consider the following code:

```
//List of integers
std::vector<int> ListOfIntegers;

//List of strings
std::vector<std::string> ListOfStrings;

//List of pointers
std::vector<CMyClass*> ListOfPointers;
```

4.3.2.3 Adding Items to a List Using `std::vector`

`std::vector` maintains a list of items, and items are added to the list at run time using the `push_back` method. This method accepts as an argument the template object to be added to the list declared as being of a matching type. The following code illustrates the adding process:

```
//List of strings
std::vector<std::string> ListOfStrings;

//Add strings to vector list
ListOfStrings.push_back("hello");
ListOfStrings.push_back("alan");
ListOfStrings.push_back("list");
```

4.3.2.4 Cycling through Items in a List Using `std::vector`

As with elements in an array or characters in an instance of `std::string`, or any other data structure where elements are arranged sequentially in memory, the items in an instance of `std::list` can be accessed individually by using the subscript (`[]`) operator (e.g., `MyList[5]`), or by using the standard STL iterators, as follows:

```
//List of strings
std::vector<std::string> ListOfStrings;

//Add strings to vector list
ListOfStrings.push_back("hello");
ListOfStrings.push_back("alan");
ListOfStrings.push_back("list");

for(int i=0; i< ListOfStrings.size(); i++)
cout<<ListOfStrings[i]<<"\n";
```

Or:

```
//List of strings
std::vector<std::string> ListOfStrings;
std::vector<std::string>::iterator myStringVectorIterator;

//Add strings to vector list
ListOfStrings.push_back("hello");
ListOfStrings.push_back("alan");
ListOfStrings.push_back("list");

for(myStringVectorIterator = ListOfStrings.begin();
    myStringVectorIterator != ListOfStrings.end();
    myStringVectorIterator++)
{
    cout<<(*myStringVectorIterator)<<"\n";
}
```

4.3.2.5 Removing Items from a List Using `std::vector`

The `std::vector` class supports both the addition of new items to the list and the removal of existing items from the list. An item (or a range of items) can be removed from the list using the `erase` method of `std::vector`, a method that accepts two STL iterator arguments specifying the start and end range of items to be deleted. The first argument is an iterator marking the first item in a range to be removed, and the second argument is an iterator marking the final item in a range of items to be removed. The following code illustrates the typical usage of the `erase` method of `std::vector` for removing items from a list.

```
//List of strings
std::vector<std::string> ListOfStrings;
```

```
//[...] add stuff to the list here

//Remove items 3-5
ListOfStrings.erase(ListOfStrings.begin()+3,
ListOfStrings.begin()+5);
```

4.4 The Game Loop

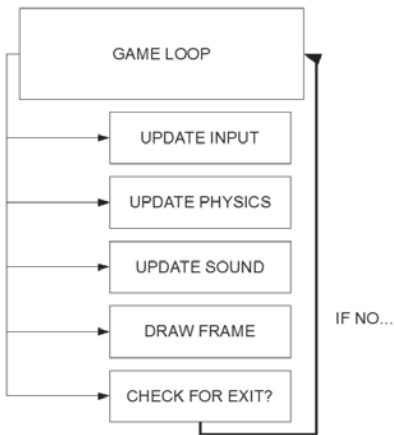


Figure 4-7

Programmatically, the dividing factor separating games from non-game software is the presence of a game loop; game software is driven by a game loop while non-game software is instead event driven. The game loop is the heartbeat (or the message pump) unique to games. As discussed earlier, non-game software such as word processors are event driven. That is, the application waits for input from the user before performing an action. For example, the user clicks the Print button and a document prints. Games differ from this event-driven arrangement, however. Certainly, games *do* respond to events. For example, in a side-scrolling platformer where the gamer must guide a character safely through a level by running and jumping across risk-laden platforms, the gamer may press the Jump button and in response the character jumps or may press the Fire button and consequently the character attacks nearby enemies. But more than this, games also work when the user does nothing; the enemies continue moving and

the game world still ticks over even when the player offers no input. In other words, the game does not freeze when user input stops. The game continues working whether or not the user is taking part; and it is this “self-directed” behavior the game loop is designed to offer. The following code is a sample C++ source file featuring a game loop set up and ready for a developer to make a game.

```
MSG mssg;

// prime the message structure
PeekMessage(&mssg, NULL, 0, 0, PM_NOREMOVE);

// run until completed
while (mssg.message!=WM_QUIT) {

    // is there a message to process?
    if (PeekMessage(&mssg, NULL, 0, 0, PM_REMOVE)) {

        // dispatch the message
        TranslateMessage(&mssg);
        DispatchMessage(&mssg);

    } else {

        //FRAME BEGINS HERE
        ReadInput();
        UpdatePhysics();
        UpdateSound();
        DrawFrame();
    }
}
```

As mentioned, the game loop is the heartbeat (or the message pump) unique to games; the loop begins after the game is executed, and exiting from the loop signals a game’s termination. In short, the game loop is a C++ while loop where each iteration (cycle) of the loop corresponds to a single frame; that is, a snapshot moment in the timeline of a game. On each iteration of the loop (on each frame) a game should:

1. **Read user input from the keyboard and mouse** to determine whether the user has moved the game character, clicked a menu item, performed any other action, or requested to exit the game (whereupon the loop should be terminated).

2. **Update game physics** based on user input and position of other game objects in the world. This may include applying gravity to objects in the air, moving the player character across the screen in the direction corresponding to an arrow keypress, etc.
3. **Update sound** to play appropriate sounds for that moment of the game, such as walking noises corresponding to player movement.
4. **Draw the frame** according to the position and perspective of the game camera in the game world. This is the final phase of the frame, the moment when all game graphics are refreshed and drawn anew to the window. We'll consider the drawing of game graphics to the application window using a third-party game development library called the SDL in the next chapter.



NOTE. The game loop is also considered in more detail in the next chapter.

4.5 Conclusion

In summary, this chapter has considered the basics of game programming in terms of three key developmental issues:

- Standardization of data types so that each game made by the same developer handles data (integers, floats, and especially strings) similarly across varied platforms
- Common framework of classes and functions to provide a cross-platform foundation upon which games can be built
- The game loop to keep a game application “alive and running,” and to configure games with a frame-based configuration rather than the event-based configuration of most non-game applications so that games may continue to operate even when no input is received or processed from the user.

The next chapter considers cross-platform game graphics programming using the SDL.

Chapter 5

SDL Graphics

This chapter focuses on developing cross-platform games that feature 2D graphics using a free, open-source software development kit (SDK) called Simple DirectMedia Layer (SDL). A software development kit is a package containing an abundance of ready-made tools, libraries, source code, and other utilities a programmer needs for making software, and the SDL is one such package designed specifically for creating cross-platform games. SDL was created in 1998 primarily by Sam Lantinga while working for California-based software firm Loki Software, which closed in 2002. Lantinga now works for game developer Blizzard Software, maker of the RPG World of Warcraft. Having been used in the creation of games such as Quake 4, Neverwinter Nights, and The Battle for Wesnoth, the SDL is arguably the SDK of choice alongside ClanLib for creating 2D games on multiple platforms.

SDL was developed using the C language but also works natively with C++ and a host of other languages including Ada, C#, Eiffel, Erlang, Euphoria, Guile, Haskell, Java, Lisp, Lua, ML, Objective C, Pascal, Perl, PHP, Pike, Pliant, Python, Ruby, and Smalltalk. Furthermore, as of 2007, the SDL development kit claims officially to support the following platforms (that is, SDL games can be compiled to run on the following operating systems): Linux, Windows, Windows CE, BeOS, Mac OS, Mac OS X, FreeBSD, NetBSD, OpenBSD, BSD/OS, Solaris, IRIX, and QNX; and it is also said to unofficially compile on the following platforms: AmigaOS, Dreamcast, Atari, AIX, OSF/Tru64, RISC OS, SymbianOS, and OS/2. The SDL is now considered in more detail.

5.1 SDL Breakdown

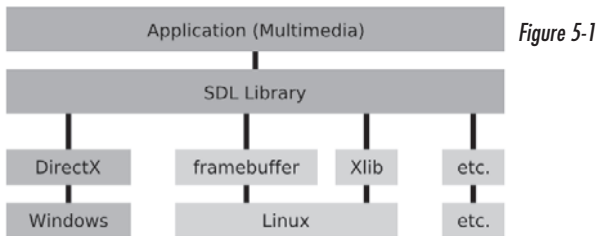


Figure 5-1

Free software and open-source, SDL is a C-based cross-platform game development kit supporting a variety of target platforms ranging from Windows to Linux. Though most developers use the SDL only in its capacity for programming game graphics, it is a multi-functional library composed of a total of eight subsystems that together go beyond game graphics, with each subsystem offering tools for developing a specific facet of a game. The eight subsystems of SDL do not need to be downloaded separately, but are all included together as one package, and are as follows:

- **Audio** — Subsystem to play sound and music in WAV format; supports both 8-bit and 16-bit, signed and unsigned samples. SDL audio support is considered later in this book. The alternative open-source and free software SDK OpenAL (Open Audio Library) is also a popular choice.
- **CD-ROM** — Subsystem to manage CD-ROM/DVD drives connected to the target PC. It includes functions to count the number of attached CD-ROMS, play CD audio tracks, and open and close the CD tray.
- **Event Handling** — Subsystem to handle and respond to common game events as they occur at run time. This includes window events (closing, minimizing), input events (mouse down, keypress), and others.

- **File I/O** — Subsystem to read from and write to files on persistent storage, including hard disks and USB memory sticks. Can be used to save and load game states and settings.
- **Joystick Handling** — Subsystem to read input from gaming input devices other than a keyboard and mouse, such as joysticks, game pads, and other similar devices, though force feedback is not supported currently. This subsystem can count the available joysticks attached to the system, read axis and button data, and read relative trackball motion.
- **Threading** — Subsystem to execute individual game processes — such as AI, player input, and audio — simultaneously using threads. This subsystem is not detailed further in this book.
- **Timers** — Subsystem to set periodical function calls and time delays, which after expiration triggers an event; in other words, a subsystem to run functions either periodically in timed intervals or only once after a specified time has elapsed. One among thousands of potential uses could be a time-delay bomb in a game that explodes after 60 seconds, and on every second until that time (every interval) a visible seconds counter decreases, counting down from 60 to 0.
- **Video** — Subsystem to draw 2D graphics such as BMPs and other animations to the display. Video does not refer specifically to motion pictures or movies such as MPEG or DVD, but more broadly to any visual image that can be presented to the screen via the system's graphics adapter. This subsystem includes functions to load images from files on disk to system memory, and to draw (*blit*) images from system memory to the screen. The video subsystem of SDL is the primary, but by no means exclusive, focus of this chapter.

5.2 Downloading and Configuring SDL

SDL is open-source and free software distributed under the GNU LGPL version 2 license, free for both commercial and non-commercial purposes, and is furthermore compliant with the Code::Blocks C++ IDE environment, as explained in an earlier chapter. This section explains how SDL can be downloaded and configured for both Windows and Linux Ubuntu.

5.2.1 SDL on Ubuntu

SDL can be downloaded and installed on Ubuntu, ready to use, in one process from the Software Repository using the Synaptic Package Manager. The following instructions illustrate the installation process step by step.

5.2.1.1 Downloading and Installing SDL on Ubuntu Using Synaptic Package Manager

1. Beginning from the Ubuntu desktop, select **System | Administration | Synaptic Package Manager**.

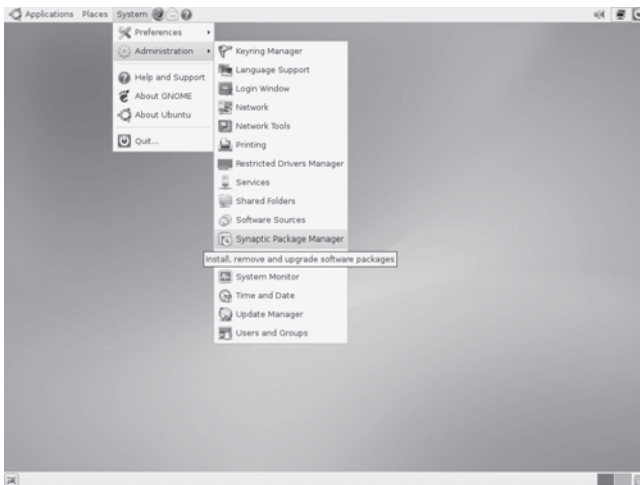


Figure 5-2

- Click **Search**, enter **SDL** to search for all available SDL-related packages, and click **Search**.

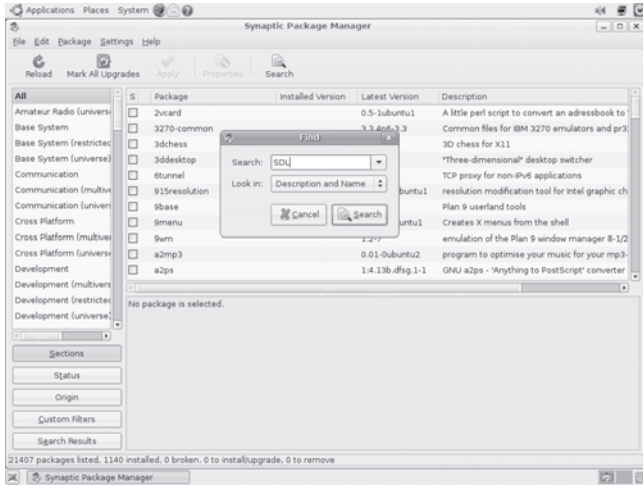


Figure 5-3

- Mark at least the `libsdl-dev` package for installation and click **Apply**. Once SDL is installed it can also be updated automatically as updates become available.

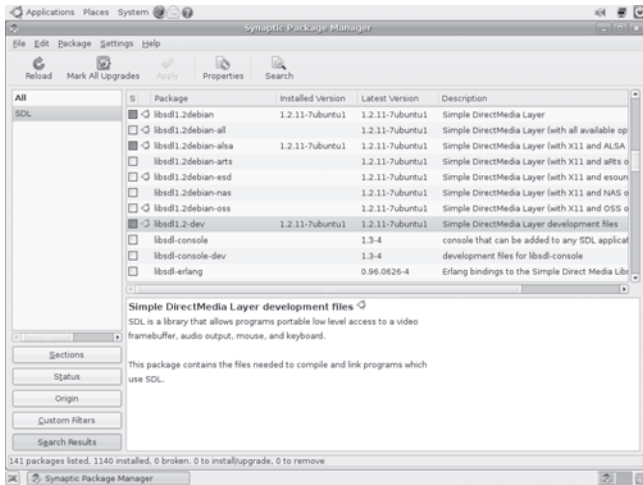


Figure 5-4

5.2.1.2 Downloading SDL Documentation from the Web

1. Go to the SDL web site at <http://www.libsdl.org/>.

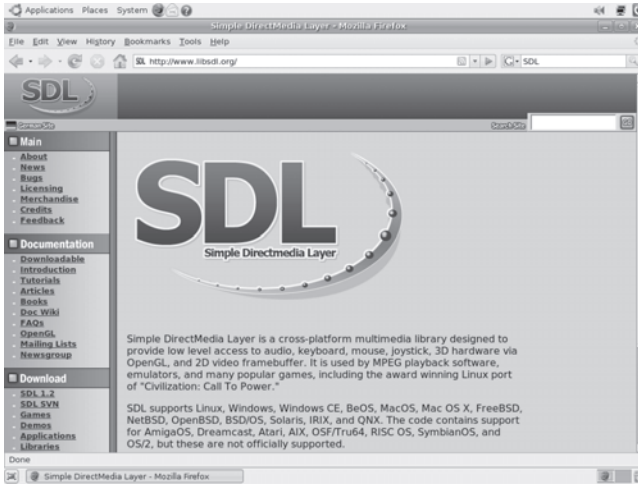


Figure 5-5

2. Under the heading Documentation, click **Downloadable**. Select the HTML SDL documentation and click **OK**. This package contains the SDL reference material that SDL programmers may need.

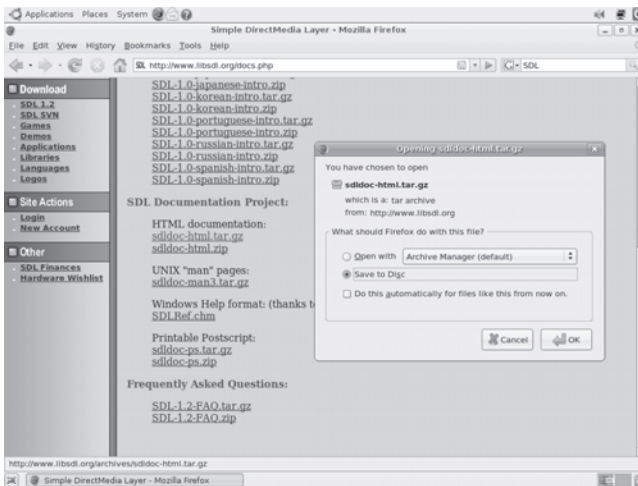


Figure 5-6

5.2.1.3 Creating an SDL Project Using Code::Blocks in Linux Ubuntu

1. Start Code::Blocks, and click the **Create New Project** option from the startup screen. A wizard appears.
2. Select the **SDL project** icon from the categories list view and click **Go**.

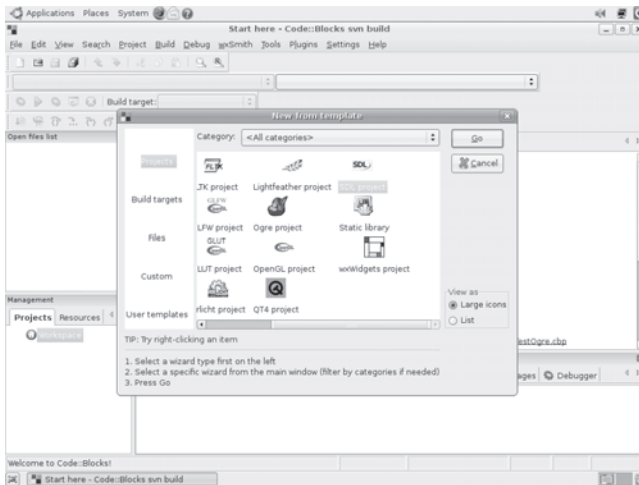


Figure 5-7

3. Click **Next** if the welcome screen appears.

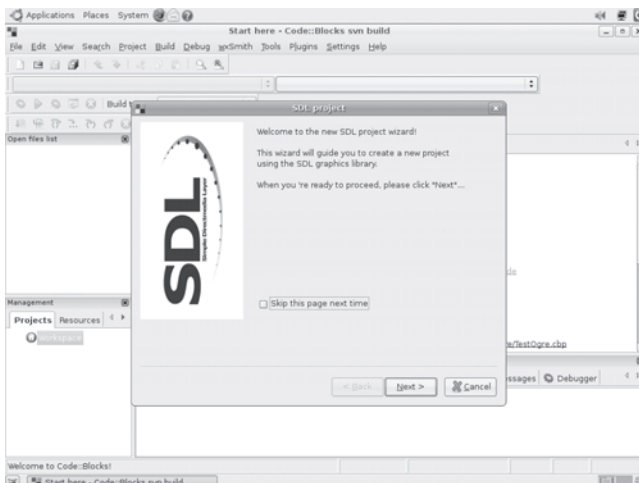


Figure 5-8

4. Enter the project details and specify the SDL project name and the fully qualified path where the source files are to be created. Click **Next**.

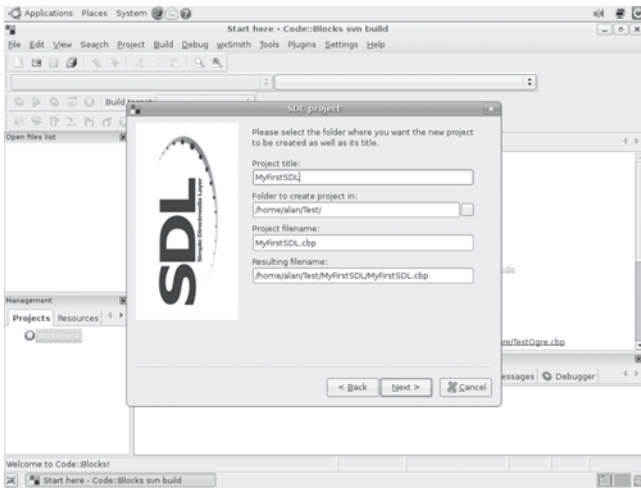


Figure 5-9

5. Select the GNU GCC Compiler from the drop-down list and click **Next**. The project is now created, with source files at the destination specified, and the project configured to use the selected compiler and linker settings.

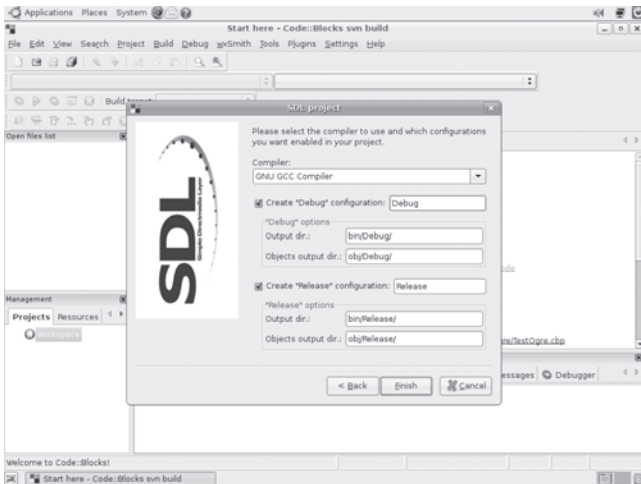


Figure 5-10

- The Code::Blocks IDE presents the source files. Click the **Compile and Run** icon to execute the project.

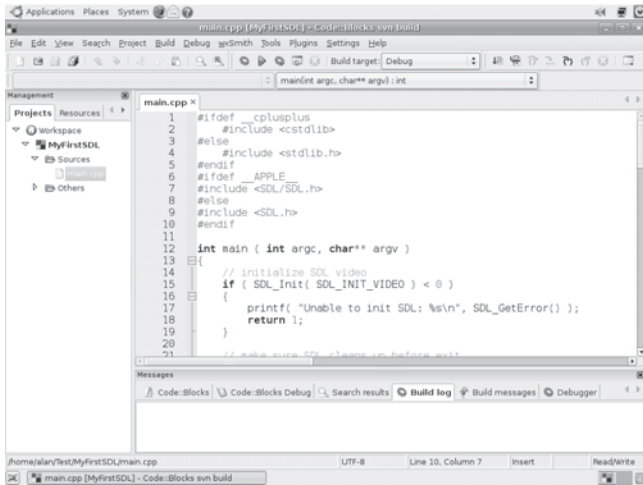


Figure 5-11

The compiled project now runs in an SDL window on Linux Ubuntu.

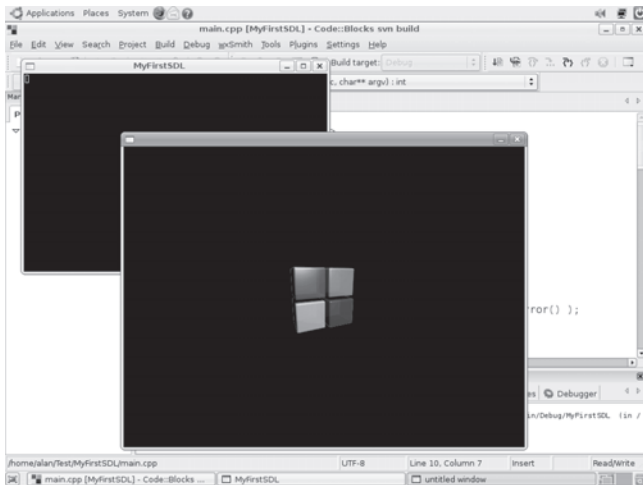


Figure 5-12

5.2.2 SDL on Windows

SDL can be downloaded and configured to compile with Code::Blocks on almost all versions of Windows available. The following instructions illustrate the installation process step by step.



TIP. Before downloading and installing SDL, users should have a means to extract TAR and 7Z compressed archives, such as by using the WinRAR or 7-Zip applications that were discussed in Chapter 3.

5.2.2.1 Downloading and Installing SDL on Windows

1. Go to the SDL web site at <http://www.libsdl.org/>.

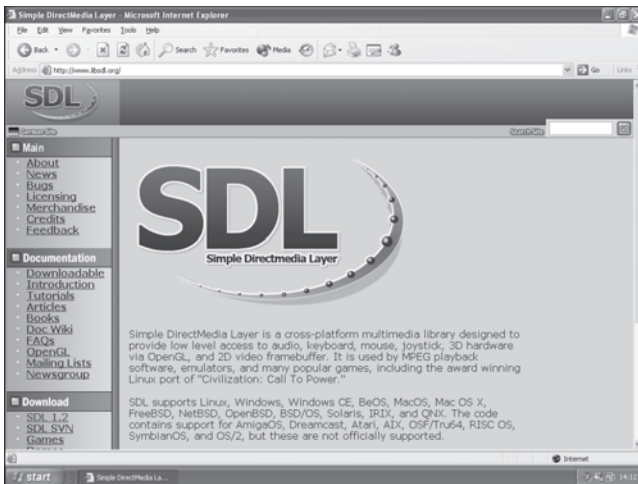


Figure 5-13

2. Under the heading Download, click the latest SDL version (SDL 1.2 at the time of writing). Download the Windows SDL runtime library `SDL-1.2.12-win32.zip`. This archive contains the important runtime `SDL.dll` file, which should be extracted to the folder from which the compiled EXE will be run. This DLL file is an SDL dependency, and since it is a “runtime” library, it is required by an SDL EXE at run time in order to execute successfully. This file must furthermore be present on all end-user machines. In other words, this file is unique among SDL files since it should be distributed to users alongside the SDL-powered game itself.

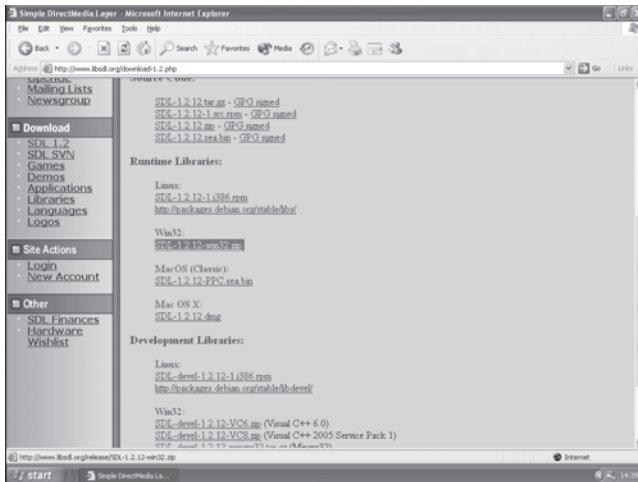


Figure 5-14

- In addition to the `SDL.dll` runtime library, SDL developers also require the SDL development libraries in order to compile SDL applications. Click the `SDL-devel-1.2.12-mingw32.tar.gz` (Mingw32) library. This archive contains the headers, source files, and libraries required for development, and these files are compliant with Code::Blocks and the GCC compiler. This archive can be extracted to any local folder on the development system.

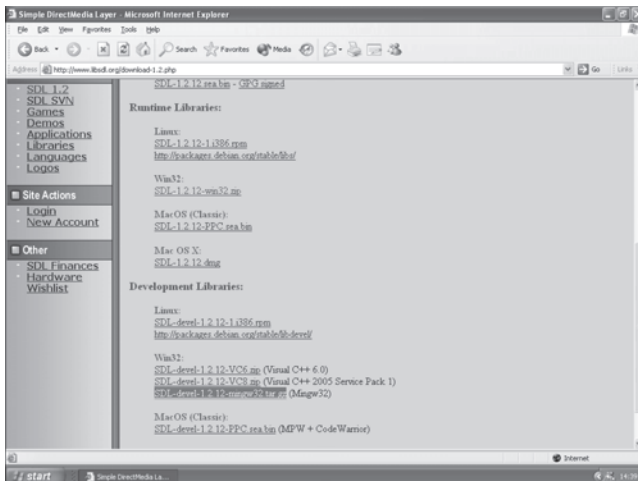


Figure 5-15



NOTE. Alternative development libraries can be downloaded for other IDEs, such as Visual Studio .NET. However, this book does not explore this IDE.

4. Scroll down the list to the SDL Documentation Project section and click the `SDLRef.chm` link to download the Windows SDL Programmer's Reference, listing all the classes, functions, and structures of SDL.

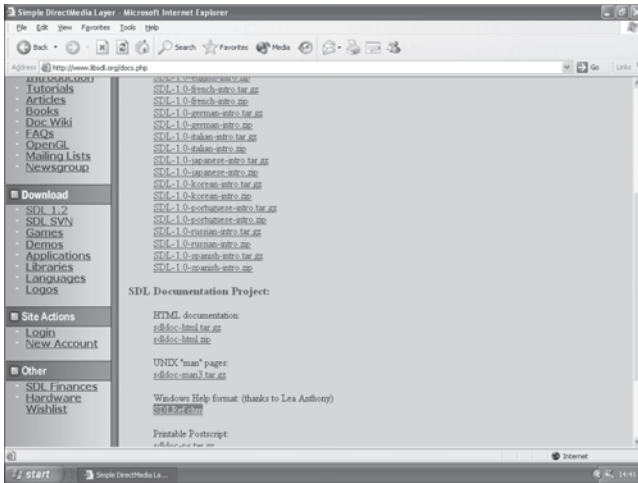


Figure 5-16

5.2.2.2 Creating an SDL Project Using Code::Blocks in Windows

1. Start Code::Blocks and click **Create New Project** from the startup screen. A wizard appears.
2. Select the **SDL project** template from the project categories list box. Click **Go**.

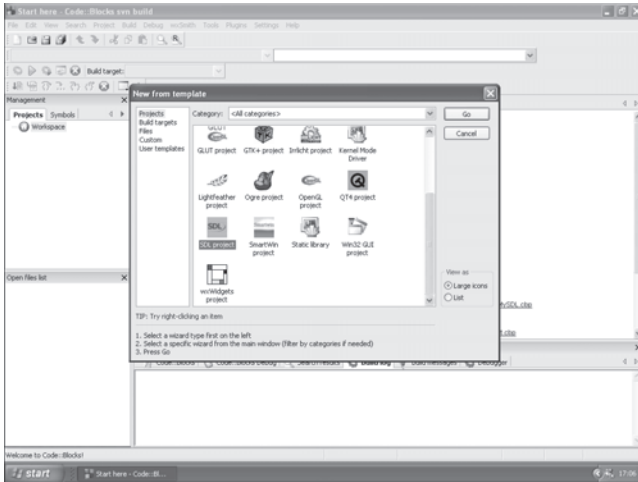


Figure 5-17

3. Click **Next** when the SDL project wizard welcome screen appears. (This welcome screen can be disabled by choosing the Skip this page next time check box.)

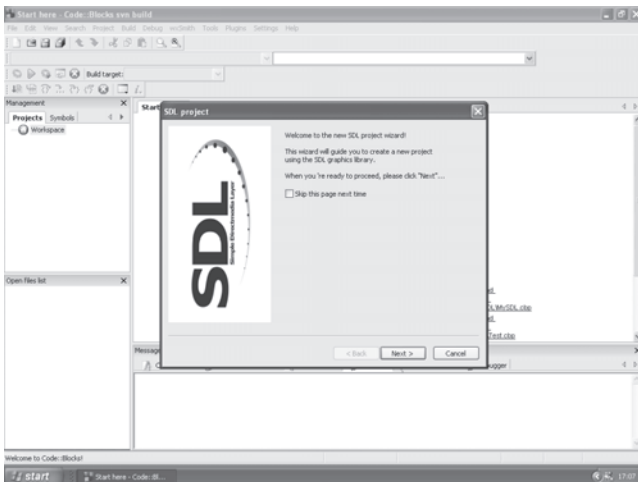


Figure 5-18

4. Assign the SDL project a name and specify a fully qualified path where the Code::Blocks project and source files will be generated. Click **Next**.

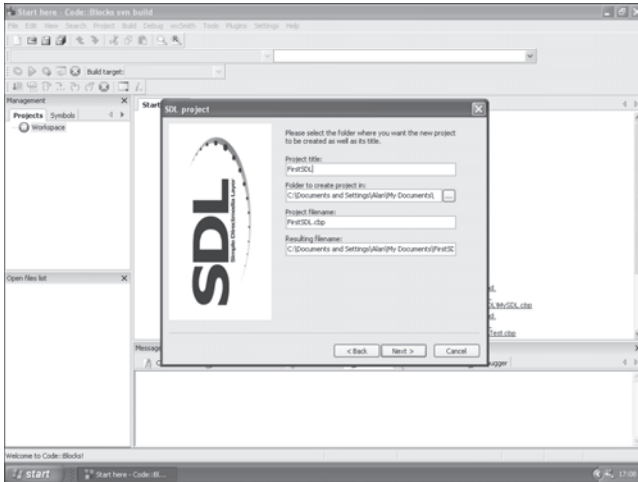


Figure 5-19

5. Specify the fully qualified path (root folder) where the SDL development files were extracted. Click **Next**.

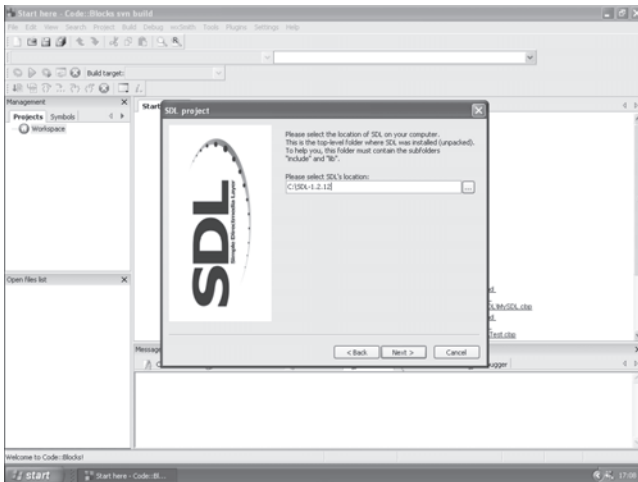


Figure 5-20

The SDL project compiles and executes in a newly created SDL window.

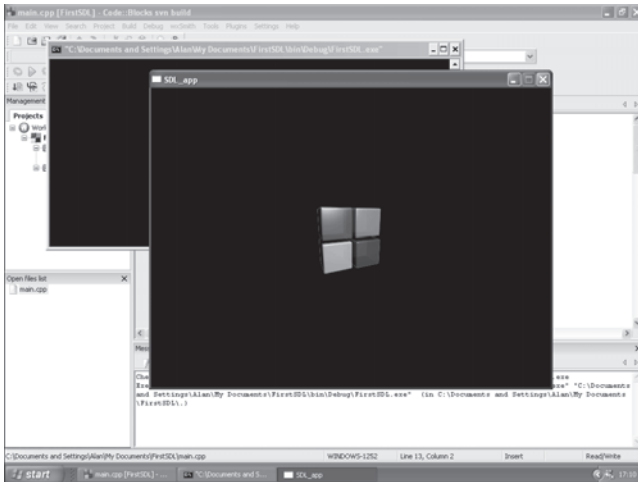


Figure 5-23



TIP. Remember, if an SDL project compiles but fails to run because of a missing `SDL.dll`, ensure the runtime library `SDL.dll` is located in the same folder in which the executable is run. Alternatively, it can be located in the Windows System32 folder.

5.3 Getting Started with SDL

SDL projects generated using the Code::Blocks application wizard are complete with code and comments, and are preconfigured to compile and run immediately. SDL applications compiled from this template feature a window inside which a bitmap is drawn on each frame during the game loop, and so the template SDL project demonstrates clearly the basic structure and most common function calls of the graphics subsystem for any SDL application. This section explores the code contained in the template project by examining the essential SDL function calls to create and sustain a window, and then explaining how images are both loaded from files to memory and drawn from memory to the window.

5.3.1 Initializing and Closing SDL

Initializing SDL is the one-time process (once per application) of preparing one or more of the SDL subsystems (audio, video, input, etc.) for use, and this process occurs through a single function call — `SDL_Init`. For this reason, initialization is the first and essential step of any SDL application and must precede all other SDL function calls. The process of *initialization* is partnered with an *uninitialization* or closing call to SDL — `SDL_Quit` — which occurs typically, though not always, at the end of an SDL application, and is the last and essential function call confirming that SDL is no longer to be used in this instance. Thus, for each call to `SDL_Init` there must be a corresponding call to `SDL_Quit`, one call marking the beginning and the other the ending of an SDL application, and all other SDL calls must occur between these two if they are to be successful.

`SDL_Init` is the function used to initialize one or more of the SDL subsystems, and takes the following form:

```
int SDL_Init(Uint32 flags);
```

- **Uint32 flags** — Unsigned 32-bit integer flag, encoding the combination of SDL subsystems to initialize. This flag should specify at least one subsystem, or can contain any combination of any of the following seven SDL subsystems (please see Section 5.1 for a more detailed description of the SDL subsystems):

```
SDL_INIT_TIMER  
SDL_INIT_AUDIO  
SDL_INIT_VIDEO  
SDL_INIT_CDROM  
SDL_INIT_JOYSTICK  
SDL_INIT_EVENTTHREAD  
SDL_INIT EVERYTHING
```



NOTE. SDL Header Files and Libraries. SDL projects generated from the Code::Blocks SDL template via the application wizard are configured to run automatically, but manually created SDL projects should include the SDL header files, as follows:

```
#include <SDL.h>  
//Or #include "MyPath/SDL.h"
```

Projects should also include the `SDLMain` library file.

Example SDL Application to Initialize and Uninitialize

```
//Initialize all SDL subsystems
if( SDL_Init( SDL_INIT_EVERYTHING ) == -1 )
{
    return 1; //Error occurred;
}
else
{
    //Success
    SDL_Quit();
}
```

5.3.2 Creating a Window and Game Loop

SDL games execute inside a window, the SDL window, and survive no longer than the window itself, which survives until the end of the application. Game loops, as described in Chapter 4, sustain the life of an application through repetition, looping to postpone the end of an application until the user terminates the loop by finally choosing to exit. Each iteration of the loop corresponds to a unique frame, one among a long sequence of frames occurring at intervals of milliseconds throughout game execution, and on each frame a programmer typically updates the game's data (player position, reading input, etc.) and also redraws the scene's graphics inside the SDL window, overwriting the pixels drawn there during the previous frame. Please consult Chapter 4 for more information on game loops and their relation to the lifetime of games specifically.

Windows are the canvas and bordered region onto which SDL games paint their graphics on each frame. More than this, however, windows are the focus for receiving input from the user and for receiving messages from the operating system generally. The SDL library offers to developers the `SetVideoMode` function to create SDL windows of a specified size, and the window lifetime is sustained by the game loop. If `SetVideoMode` is successful, the function returns a pointer to an instance of `SDL_Surface`, which is a class encapsulating an active window that has properties and methods to set the window's title and size, among other features. `SetVideoMode` is detailed below.

```
SDL_Surface *SDL_SetVideoMode(int width, int height, int bpp,  
                               Uint32 flags);
```

- **int width, int height** — Integer parameters to specify the width and height in pixels of the window to be created.
- **int bpp** — Specifies the bits per pixel of the window to be created; usually 16.
- **Uint32 flags** — Unsigned 32-bit integer flag specifying additional properties of the window to be created. This flag may be a combination of one or more of the following values:
 - **SDL_SWSURFACE** — The SDL window and its contents are allocated to system memory, not to the memory of video hardware. Greater system compatibility, worse performance generally.
 - **SDL_HWSURFACE** — The SDL window and its contents are allocated to hardware video memory, not to system memory. Less system compatibility, better performance generally.
 - **SDL_FULLSCREEN** — This flag creates a full-screen SDL window.



NOTE. The code listed in bold refers to particularly important or key functions.

Example SDL Application

In this sample code we create a window and sustain its lifetime using a game loop.

```
// create a new window  
SDL_Surface* screen = SDL_SetVideoMode(640, 480, 16,  
                                        SDL_HWSURFACE|SDL_DOUBLEBUF);  
  
if ( !screen )  
{  
    printf("Unable to set 640x480 video: %s\n", SDL_GetError());  
    return 1;  
}  
  
bool done = false;  
while (!done)  
{
```

```
// message processing loop
SDL_Event event;
while (SDL_PollEvent(&event))
{
    // check for messages
    switch (event.type)
    {
        // exit if the window is closed
        case SDL_QUIT:
            done = true;
            break;

        // check for keypresses
        case SDL_KEYDOWN:
            {
                // exit if ESCAPE is pressed
                if (event.key.keysym.sym == SDLK_ESCAPE)
                    done = true;
                break;
            }
    } // end switch
} // end of message processing

// DRAWING STARTS HERE

// clear screen
SDL_FillRect(NULL, 0, SDL_MapRGB(screen->format, 0, 0, 0));

// GAME UPDATE AND DRAWING OCCURS HERE
//Do stuff
// DRAWING ENDS HERE

// Update the screen
SDL_Flip(screen);

} // end main loop
```

The SDL code sample featured above is a classic example of an SDL application insofar as it creates and maintains a 640 x 480 window using a game loop, a loop that also reads and processes user input to determine when the Esc key on the keyboard is pressed to signal the end of the loop. In short, the above code sample illustrates at least

three crucial components and subsystems at work in SDL: surfaces, event polling, and page flipping.

- **Surfaces** — The SDL video subsystem may potentially contain many surfaces in memory at run time, at least one of which is returned from the `SetVideoMode` function and represents the pixel contents of the SDL window. An SDL surface represents a readable and writable canvas of pixels in memory; that is, a rectangular region of bytes that may be loaded with image data from files on disk or from other surfaces by copying and pasting. A single instance of `SDL_Surface` corresponds to a single surface in memory, offering methods and properties for accessing and editing the pixels on the surface. Surface handling is considered in more detail shortly.
- **Event polling** — The SDL event subsystem handles all system events. The term “event” refers to the run-time notification sent to an application as an important situation occurs, such as a keypress, mouse movement, or system error. So, the SDL event subsystem handles all events automatically as they occur by posting (or adding) them onto an event queue (or event stack). The events in the event queue accumulate unprocessed one atop the other until the SDL application requests, or *polls* for, the events using the `SDL_PollEvent` function, responding to each event appropriately. On each frame, or iteration of the game loop, an application polls for all pending events using the `SDL_PollEvent` function. This function removes and returns only the topmost event on the stack, and so must be called as many times as there are pending events in order to empty the queue on each frame. Returned events are encoded in an `SDL_Event` structure, as follows:

```
typedef union{
    Uint8 type;
    SDL_ActiveEvent active;
    SDL_KeyboardEvent key;
    SDL_MouseMotionEvent motion;
    SDL_MouseButtonEvent button;
    SDL_JoyAxisEvent jaxis;
    SDL_JoyBallEvent jball;
    SDL_JoyHatEvent jhat;
    SDL_JoyButtonEvent jbutton;
```

```
SDL_ResizeEvent resize;
SDL_ExposeEvent expose;
SDL_QuitEvent quit;
SDL_UserEvent user;
SDL_SywmEvent syswm;
} SDL_Event;
```



TIP. The `SDL_PollEvent` returns 0 if the event queue is empty; in other words, when there are no pending events to handle.

Example SDL Event Loop

```
while (SDL_PollEvent(&event))
{
    // check for messages
    switch (event.type)
    {
        // exit if the window is closed
        case SDL_QUIT:
            done = true;
            break;
    }
}
```



NOTE. Appendix G at the end of this book features a table listing all of the SDL key code constants.

- **Page flipping** — SDL applications typically refresh, or repaint, the graphical contents of their window on each frame, and since a frame corresponds to an iteration of the game loop, an application must repaint the window on each iteration. SDL repaints the main window instantaneously whenever a call to `SDL_Flip` is made, and repainting occurs during this function via a process called *page flipping*. To repaint in this instance means to update the pixel data inside the main window, revealing the contents of whatever has been drawn there since the previous call to `SDL_Flip`.
-

Page Flipping

The term *page flipping*, also known as *double buffering*, refers to a flip book style technique for drawing computer graphics to the display quickly and efficiently. Software that uses page flipping will traditionally maintain at least two canvases (or *surfaces*; regions for drawing pixel data in memory), one surface being read-only and visible (on-screen) and the other being read/write and hidden (off-screen). That is, the on-screen surface is a final composition, and the off-screen surface is a work in progress free to be edited and painted to. Page flipping occurs at the end of each frame, when painting to the off-screen surface is completed. It is a process of flipping, or exchanging the on-screen and off-screen surfaces, such that the off-screen work in progress now becomes the on-screen and read-only surface, and the previously on-screen surface now becomes an off-screen work in progress. Thus, the surfaces are switched, and the off-screen surface becomes visible, and so the process goes on for each frame. In this way, games designate a surface for working and a surface for presenting the latest contents drawn on each frame.

5.3.3 SDL Surfaces

As stated earlier, the SDL video subsystem is responsible chiefly for the graphics of an SDL application and may potentially contain many *surfaces* at run time, at least one of which is returned from the `SetVideoMode` function and represents the pixel contents (the canvas) of the SDL window. However, programmers may create other surfaces manually in addition to the SDL window by using a variety of SDL functions, such as the `SDL_LoadBMP` function. To clarify, an SDL surface refers to a readable and writable canvas of pixels in memory. The SDL function `SDL_LoadBMP` creates an SDL surface in memory loaded with pixel data from a valid bitmap file on disk. `SDL_LoadBMP` takes the following form:

```
SDL_Surface *SDL_LoadBMP(const char *file);
```

- **const char *file** — Fully qualified path of the file whose pixel data will be loaded onto the newly created surface returned by the `SDL_LoadBMP` function.



NOTE. `SDL_LoadBMP` creates a new SDL surface in system or hardware memory that is loaded with pixel data from a bitmap file. So each call to `SDL_LoadBMP` should be later followed by a matching call to `SDL_FreeSurface` (when a surface is no longer required by an application), deleting the created surface from memory.

Example SDL Surface Loaded from a File

```
SDL_Surface* loadedImage = NULL;
loadedImage = SDL_LoadBMP(filename.c_str());
```

5.3.3.1 Blitting Surfaces

The process of copying pixels from one surface to another is called *blitting*, and the SDL video subsystem offers the `SDL_BlitSurface` function to blit pixels between surfaces loaded in memory, just as pixels can be copied and pasted between surfaces in a photo editing package like Photoshop. In short, blitting means a rectangle of pixels is copied from a source surface and pasted onto a destination. Note that the main window surface created by `SetVideoMode` is unique among SDL surfaces since it singularly represents the off-screen buffer in a page flipping chain. That is, the pixel contents of the window surface become visible when the `SDL_Flip` function is called at the end of each frame. Thus, SDL applications can present off-screen surfaces loaded from files to the display by blitting them to the main window surface on each iteration of the game loop. In other words, surfaces are made visible when blitted to the window surface. The `SDL_BlitSurface` function used to blit one surface to another takes the following form:

```
int SDL_BlitSurface(SDL_Surface *src, SDL_Rect *srcrect,
                   SDL_Surface *dst, SDL_Rect *dstrect);
```

- **SDL_Surface *src** — Pointer to a valid instance of `SDL_Surface` that is the source of the blit operation.
 - **SDL_Rect *srcrect** — Pointer to a valid `SDL_Rect` structure specifying the rectangular subset of pixels to be copied from the source surface, specified by `src`. `NULL` selects the entire surface.
-

- **SDL_Surface *dst** — Pointer to a valid instance of `SDL_Surface` that is the destination of the blit operation.
- **SDL_Rect *dstrect** — Pointer to a valid `SDL_Rect` structure specifying the rectangular subset of pixels into which the source pixels from the source surface should be pasted.

Example SDL Surface Blitted to a Window

```
//Blits a bitmap surface onto the main window canvas
SDL_BlitSurface(source, NULL, screen, NULL);
```

5.3.3.2 Optimizing SDL Surfaces

Surfaces are said to be *optimized* when their pixel format (bits per pixel) matches the pixel format of their application’s main window, also known as the “main surface” or the “frame buffer.” SDL surfaces created by the video subsystem from bitmap files on disk are usually created with a pixel format matching the format of the file from which they were loaded, such as 16-bit surfaces from 16-bit files. However the format of a surface, since it may come from one among potentially many image files, may not match that of the frame buffer onto which most surfaces are ultimately copied (*blitted*) on each frame for display. (Pixels from one surface can be copied and pasted onto another using the `SDL_BlitSurface` function, explained in the previous section.) This means that for each time a surface is copied to a destination whose pixel format differs, SDL must perform a conversion of the source pixels into the format of the destination. This conversion inevitably incurs a performance penalty for an application on every occasion where it must occur. To solve this issue, the SDL video subsystem therefore provides the `SDL_DisplayFormat` function, which accepts as an argument a pointer to any standard surface loaded from a file on disk and returns in memory a pixel-by-pixel duplicate surface whose pixel format matches the frame buffer. The `SDL_DisplayFormat` function is designed to be called only once for each non-frame buffer surface created. The prototype for `SDL_DisplayFormat` is featured below:

```
SDL_Surface *SDL_DisplayFormat(SDL_Surface *surface);
```

- **SDL_Surface *surface** — Specifies a valid instance of class `SDL_Surface`, whose duplicate with an amended pixel format matching the frame buffer is returned by `SDL_DisplayFormat`.

Example SDL Surface Loaded from a File and Optimized

```
//Load the image
SDL_Surface* loadedImage = NULL;
SDL_Surface* optimizedImage = NULL;
loadedImage = SDL_LoadBMP(filename.c_str());

if(loadedImage != NULL)
{
    //Create optimized image
    optimizedImage = SDL_DisplayFormat(loadedImage);

    //Free old image
    SDL_FreeSurface(loadedImage);

    loadedImage = NULL;
}
```



NOTE. Optimizing an SDL surface creates a duplicate surface whose pixel format matches the frame buffer. Thus, the original unoptimized surface can safely be deleted from memory using the `SDL_FreeSurface` function, as demonstrated in the code example above.

5.3.4 Additional File Formats (JPEG, PNG, TGA, and Others)

The by now familiar `SDL_LoadBMP` function of the SDL video subsystem allows programmers to create surfaces in memory with pixels loaded from bitmap files (.bmp) only. This function, and the SDL more generally, does not by default support surfaces loaded from image files other than bitmap, such as from JPEG, PNG, or TGA files. The ability to create surfaces from such alternative formats is supported only by an additional package of development libraries available from the SDL web site or the Ubuntu Repositories. These libraries are freely available and under the umbrella of the SDL development kit, and therefore are subject to the same GNU licensing terms. This section explores the step-by-step process for downloading the SDL Image Development Libraries, and further examines how to configure them for use in SDL projects created by Code::Blocks on both Windows and Linux Ubuntu.

5.3.4.1 Downloading and Configuring SDL Image Development Libraries for Code::Blocks on Ubuntu

1. Beginning from the Ubuntu desktop, select **System | Administration | Synaptic Package Manager** from the main menu.
2. Click **Search** and type **SDL** to filter the packages for a list of SDL-related development files. Mark the **libsdl-image-dev** files for installation and click **Apply**.

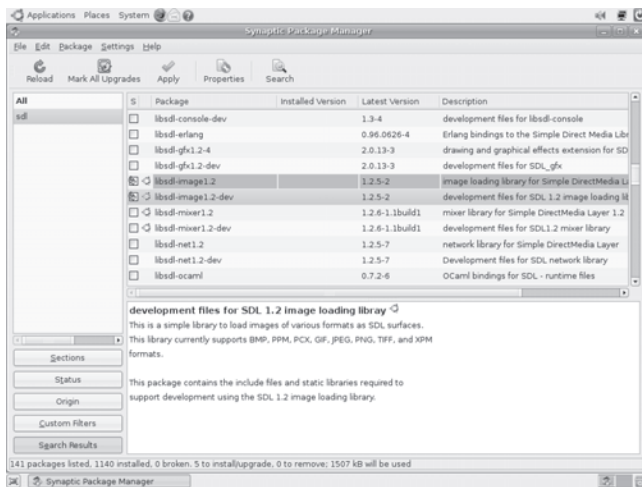


Figure 5-24

3. Once the SDL Image Development Libraries are installed, start Code::Blocks.
4. Load an existing SDL project, or create a new SDL project using the steps in Section 5.2.1.3.

- From the Code::Blocks Editor's main menu, select **Project | Build options**.

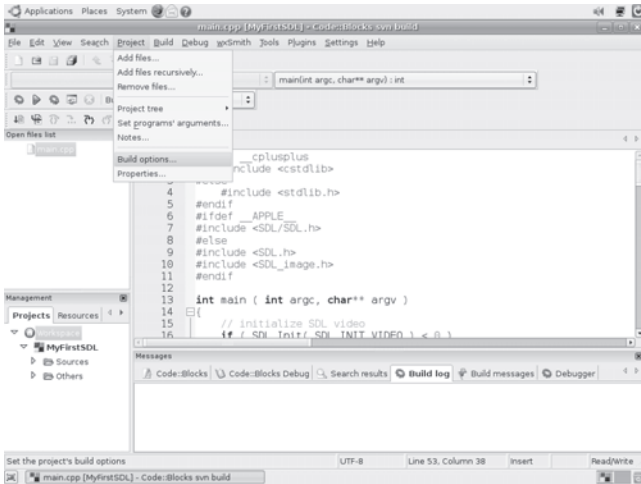


Figure 5-25

- Click the **Linker settings** tab and type **SDL_image** in the Link libraries list box. Click **OK**.

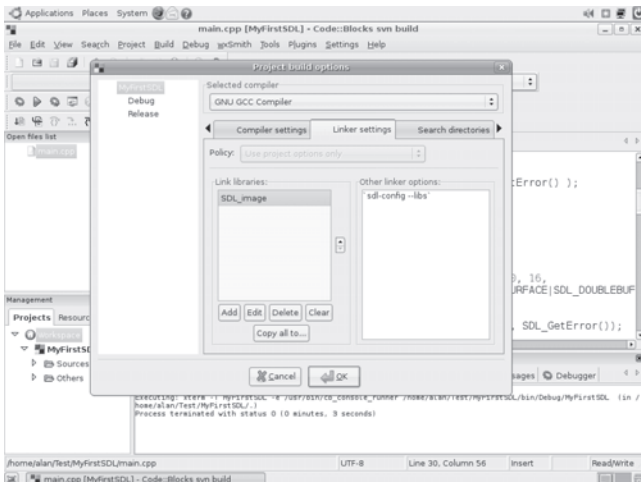


Figure 5-26

- Include the **#include <SDL_image.h>** preprocessor directive in addition to existing directives in all SDL projects to use the SDL image development functions. The project is now configured.

The next section details the Windows installation for the SDL Image Development Libraries, and a later section explains how to load SDL surfaces from images in formats other than BMP.

5.3.4.2 Downloading and Configuring SDL Image Development Libraries for Code::Blocks on Windows

1. Beginning from the Windows desktop, navigate a web browser to the SDL development libraries web site at http://www.libsdl.org/projects/SDL_image/.



NOTE. Or alternatively, go to the SDL home page at <http://www.libsdl.org/>. Under the Download heading on the left, click Libraries, then enter the search keyword Image in the edit box. Scan the results for the SDL Image Library.

2. Download the SDL Image Development package `SDL_image-devel`.

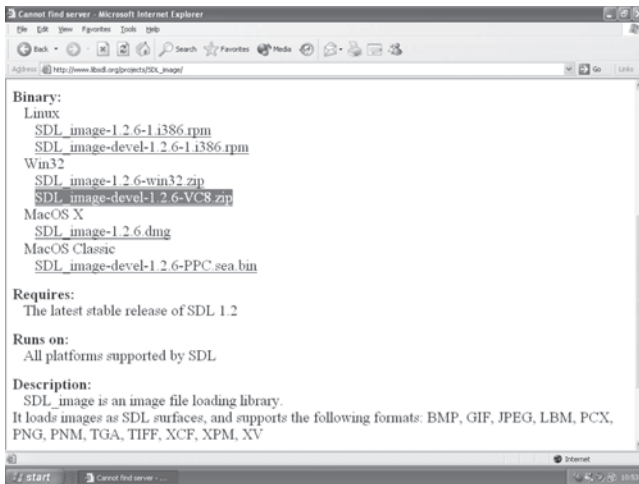


Figure 5-27

3. Once downloaded, extract the header, source, library, and DLL files of the zip package to the current SDL folder on the local machine. Then start Code::Blocks.
4. Open an existing SDL project or create a new SDL project from the Code::Blocks SDL template wizard, as described in Section 5.2.2.2.

5. From the Code::Blocks Editor's main menu, select **Project | Build options**.
6. Click the **Linker settings** tab and add the SDL Image Development Library (SDL_Image) to the Link libraries list box. Click **OK**.

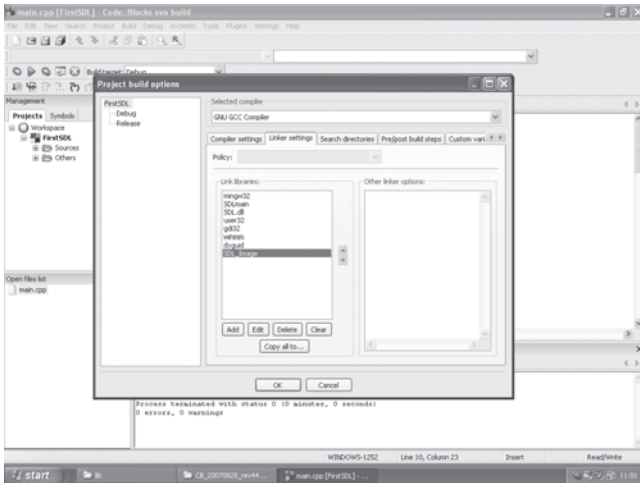


Figure 5-28

7. Include the `#include <SDL_image.h>` preprocessor directive in addition to existing directives in all SDL projects to use SDL image development functions. The project is now configured.

5.3.4.3 SDL: Further Image Formats

Now let's look at how to load SDL surfaces from images in formats other than BMP. The following code loads a PNG file.



NOTE. The code sample is similar to the sample featured earlier in this chapter where a surface was loaded from a bitmap file using `SDL_LoadBMP`, except the function `IMG_Load` has replaced a call to `SDL_LoadBMP`. Like `SDL_LoadBMP`, the function `IMG_Load` creates a standard `SDL_Surface` in memory, but it can load surface pixels from any valid image file on disk in any of the following formats in addition to BMP: GIF, JPEG, LBM, PCX, PNG, PNM, TGA, TIFF, XCF, XPM, and XV.

Example SDL Surface Loaded from a PNG File and Optimized

```
//Load the image
SDL_Surface* loadedImage = NULL;
SDL_Surface* optimizedImage = NULL;
loadedImage = IMG_Load("myfile.png");

if(loadedImage != NULL)
{
    //Create optimized image
    optimizedImage = SDL_DisplayFormat(loadedImage);
    //Free old image
    SDL_FreeSurface(loadedImage);

    loadedImage = NULL;
}
```

5.4 Color Keying with Surfaces

It has been demonstrated throughout the previous sections of this chapter how the SDL video subsystem features at least two functions, `SDL_LoadBMP` and `IMG_Load`, for creating a rectangular canvas of bytes in memory called a surface; and these functions further allow the surface to be loaded instantly with pixels from an image file on disk in a variety of formats, from BMP to TGA. Furthermore, specified rectangles of pixels may be copied and pasted (blitted) to and from surfaces in memory using the `SDL_BlitSurface` function, and pixels that are copied to the main window (main surface, or frame buffer) are drawn to the game window display and thereby become visible to the gamer for every frame they are blitted there.

Until now the `SDL_BlitSurface` function has been used to copy pixels between a source and a destination surface in memory, and a programmer does this by specifying a finite-sized rectangle of pixels on the source surface to copy onto a specified location on the destination surface. This means all pixels inside the specified source rectangle are indiscriminately blitted to the destination rectangle, regardless of their contents. There are often situations, however, when SDL applications must blit only certain pixels, such as those of a

specific color, rather than all pixels inside the rectangle. Consider the example of the two images loaded onto SDL surfaces featured in Fig-



ure 5-29 — the cityscape background and the happy face.

The first surface may be blitted entirely to the frame buffer where it will be presented to the display as a background for the scene, but the second surface (the happy face) features superfluous pixels on the outside of the face that should be removed when it is blitted to the background. Thus, the superfluous pixels around the face must be ignored when the surface is blitted to the destination, and so the SDL video subsystem must distinguish between these pixels on the basis of that which each of those pixels share, namely their color. The process of filtering, or ignoring, specified pixels from the source surface on the basis of color as it is blitted to a destination surface is known as *color keying*. The SDL video subsystem offers the `SDL_SetColorKey` function to set the color key specifically for any valid surface in memory. Only one color key may be applied to a surface at any one time, and for color keying to work successfully, the `SDL_SetColorKey` function should also be called *before* any blit operations involve the surface on which the color key is to be applied. The `SDL_SetColorKey` function takes the following form:

```
int SDL_SetColorKey(SDL_Surface *surface, Uint32 flag, Uint32
    key);
```

- **SDL_Surface *surface** — Pointer to a valid instance of `SDL_Surface` in memory, representing the surface for which the color key is to be applied.
- **Uint32 flag** — Specify `SDL_SRCCOLORKEY`.
- **Uint32 key** — Unsigned 32-bit integer specifying the color itself to be used as the color key for the surface represented by the argument surface. This color value can be generated using the

RGB (Red, Green, Blue) color composition macro, `SDL_MapRGB`, as demonstrated in the following code sample.

Example SDL Surface Loaded from a PNG File, Optimized, and a Color Key Applied

```
//Load the image
SDL_Surface* loadedImage = NULL;
SDL_Surface* optimizedImage = NULL;
loadedImage = IMG_Load("myfile.png");

if(loadedImage != NULL)
{
    //Create optimized image
    optimizedImage = SDL_DisplayFormat(loadedImage);

    //Free old image
    SDL_FreeSurface(loadedImage);
    loadedImage = NULL;

    Uint32 colorkey = SDL_MapRGB(optimizedImage->format, 0,
                                0xFF, 0xFF);

    SDL_SetColorKey(optimizedImage, SDL_SRCCOLORKEY, colorkey);
}
```

5.5 Conclusion

To summarize, the SDL (Simple DirectMedia Layer) was established in 1998 and is perhaps the foremost SDK among the open-source, cross-platform, and free software used primarily by programmers for creating video games, and more commonly for handling video game graphics. The SDL is a single library made up of eight conceptual sub-systems: audio, CD-ROM, event handling, file I/O, joystick handling, threading, timers, and video. This chapter focused narrowly on some of the fundamental details of the SDL video subsystem, an umbrella term encompassing a plethora of topics, such as page flipping, surfaces, blitting, and color keying. To conclude, the following sample SDL source code is provided, forming a complete SDL program made

with Code::Blocks, and highlighting the topics featured in this chapter — from configuring an SDL application to blitting between surfaces using color keying. The next chapter will examine in greater detail much of the SDL subject matter explained here in order to apply SDL in practical game-making techniques. (The following code is also provided in the book's companion files available at www.wordware.com/files/gamedev056X)

Example SDL Application

```
#ifdef __cplusplus
    #include <cstdlib>
#else
    #include <stdlib.h>
#endif
#ifdef __APPLE__
    #include <SDL/SDL.h>
#else
    #include <SDL.h>
    #include <SDL_image.h>
#endif

int main (int argc, char** argv)
{
    // initialize SDL video
    if (SDL_Init(SDL_INIT_VIDEO) < 0)
    {
        printf("Unable to init SDL: %s\n", SDL_GetError());
        return 1;
    }

    // make sure SDL cleans up before exit
    atexit(SDL_Quit);

    // create a new window
    SDL_Surface* screen = SDL_SetVideoMode(800, 600, 16,
                                           SDL_HWSURFACE|SDL_DOUBLEBUF);

    if ( !screen )
    {
        printf("Unable to set 640x480 video: %s\n",
              SDL_GetError());
        return 1;
    }
}
```

```
// load an image
SDL_Surface* bmp = IMG_Load("/home/alan/Desktop/Test.bmp");
SDL_Surface* optimizedImage = NULL;
if (!bmp)
{
    printf("Unable to load bitmap: %s\n", SDL_GetError());
    return 1;
}
else
{
    optimizedImage = SDL_DisplayFormat(bmp);
    SDL_FreeSurface(bmp);

    bmp = NULL;

    Uint32 colorkey = SDL_MapRGB(optimizedImage->format, 0,
                                0xFF, 0xFF);

    SDL_SetColorKey(optimizedImage, SDL_SRCCOLORKEY, colorkey);
}

// center the bitmap on screen
SDL_Rect dstrect;
dstrect.x = (screen->w - optimizedImage->w) / 2;
dstrect.y = (screen->h - optimizedImage->h) / 2;

// program main loop
bool done = false;
while (!done)
{
    // message processing loop
    SDL_Event event;
    while (SDL_PollEvent(&event))
    {
        // check for messages
        switch (event.type)
        {
            // exit if the window is closed
            case SDL_QUIT:
                done = true;
                break;
        }
    }
}
```

```
        // check for keypresses
    case SDL_KEYDOWN:
    {
        // exit if ESCAPE is pressed
        if (event.key.keysym.sym == SDLK_ESCAPE)
            done = true;
        break;
    }
    } // end switch
} // end of message processing

// DRAWING STARTS HERE

// clear screen
SDL_FillRect(screen, 0, SDL_MapRGB(screen->format, 0, 0, 0));

// draw bitmap
SDL_BlitSurface(optimizedImage, 0, screen, &dstrect);

// DRAWING ENDS HERE

// finally, update the screen :)
SDL_Flip(screen);
} // end main loop

// free loaded bitmap
SDL_FreeSurface(optimizedImage);

// all is well ;)
printf("Exited cleanly\n");
return 0;
}
```

Chapter 6

Game Audio

The adage “sound can make or break a game” has no doubt been around for as long as there have been games with sound, and it contains in one succinct message a double-edged meaning. First, it hints at the general importance of sound for a gaming experience, but secondly, and more ominously, it also suggests that “no sound” can be better than “bad” sound since bad sound can break a game. For a game developer, then, creating sound for a game is mainly a balancing act between silence and sound; when and when not to play audio in a game. This is largely the subject matter for this chapter, which focuses on programming audio using the SDL library for cross-platform games.

In the game development world, the notion of sound (or more generally, audio) is conceptually divided into two subcategories: sound and music. The distinction is drawn between the two kinds of audio based upon its purpose in a given video game. Audio categorized as sound (or *SFX*) includes the sounds played as events happen in the game world such as footstep sounds played when a game character is walking, gunshot and glass breaking sounds played during a gun battle between two opponents, speech played during a conversation, and game menu sounds played for events like button presses and the appearance of confirmation dialogs such as “Do you want to exit?” Audio categorized as background music (*BGM*) or *incidental music* is generally longer in duration than a sound; a sound may last for less than 30 seconds whereas music may play for longer than two minutes.

A sound is often played by a game repeatedly and consciously on every occasion a specified event occurs (gun shot, punch sound, scream) to *reinforce* the effect that “something” has happened, as opposed to nothing has happened; to represent action without visuals. By contrast, music is played subliminally and continuously for long periods of time to convey an atmosphere; a feeling or a mood.

Background music usually refers to an unobtrusive musical score that begins in the same way it ends so it can be looped seamlessly in the background to convey a smooth, subliminal mood for as long as the player remains in the current scene of the game. A single track of background music is said to be unobtrusive when it isn't loud, fast-paced, or packed with vocals and other attention-grabbing features that divert a player's attention away from the game and toward the music alone.

Single-track background music is designed to be subliminal and is typically assigned to play repeatedly and continuously across one game scene (one building, one room, one level, etc.) regardless of the events that take place there.

In contrast, incidental music refers to a process of changeable music; music that doesn't remain constant but changes to reflect the events occurring in the "here and now" of the current game scene. For example, in a random death-match level of an FPS game, the player may have become injured by his opponent's attack during a skirmish, and fled into the darkness of emptier parts of the level in search of cover, health, power-ups, or better weapons, or to engage in guerilla warfare. The player's retreat from battle pays off since his character finds a safe haven in which to recover from any injuries; here the music is tranquil, a short, looping background track of the "chill-out" kind, and perhaps now and then a "sighing" or "breathless" SFX played to represent the player's recovery. Then suddenly, BANG!, the enemy encroaches into the player's enclave with guns blazing and at once the player is again hoisted into the throes of battle; here the music changes, bursting into an intense beat that began with the enemy's surprise attack. This is a classic example of incidental music at work. Thus, incidental music describes a dynamic process whereby a game elects to play the background music most appropriate for the events unfolding in the current game scene.

6.1 Recording and Editing Game SFX

Those who create and edit sound effects for video games may seem peculiar and possibly disturbed people to those not familiar with the intricacies of game development. Sound effect creators can often be found in lonely corners of rooms hunched over microphones making the strangest of noises with the most unexpected of objects; recording gnarls, snorts, the office air conditioning unit, the flush of the toilet system, the sound of burping through a flexible hose, and the creaks in the floorboards to name but a few of the sounds in their audible wonderland. For them it appears nothing is off-limits and every sound exists only to be recorded. This, then, is the philosophy of sound effect creation, and it is by recording sounds using a microphone that most game developers collect their sounds for use in their games, arranging them into a database called a “sound bank.” However, recording sounds manually is not the only method a developer employs for obtaining sound effects; developers may also purchase compilation CDs and DVDs featuring sound banks containing hundreds or thousands of sounds for use in their games.

Free Sound

Recently, the free software and open-source movement has also spawned a wide variety of patent-free codecs (such as Ogg Vorbis) and “creative commons” media. The FreeSound project (<http://freesound.iaa.upf.edu/>), for example, is an online sound bank featuring sounds licensed under the Creative Commons Sampling Plus License, meaning any user can download the sound effects listed there and use them in their products. The Creative Commons License grants the following (as listed at <http://creativecommons.org/licenses/sampling+/1.0/>).

“You are free:

1. To sample, mash-up, or otherwise creatively transform this work for commercial or non-commercial purposes.

2. To perform, display, and distribute copies of this whole work for non-commercial purposes (e.g., file-sharing or non-commercial webcasting).

Under the following conditions:

- You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).
- You may not use this work to advertise for or promote anything but the work you create from it.
- For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to this web page.

Note: For reference, the full text of the Creative Commons License can be found in Appendix C at the end of this book.

6.2 SFX Software

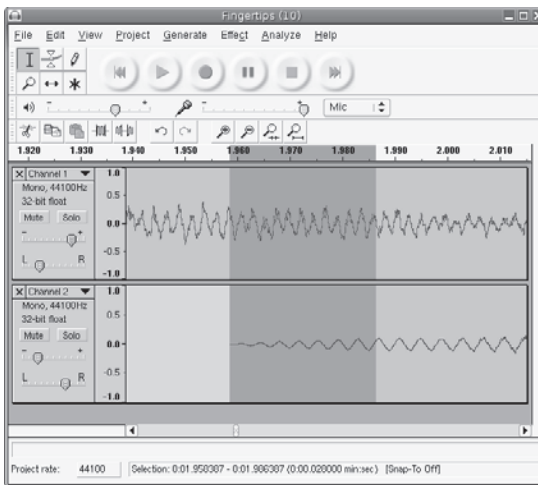


Figure 6-1: Audacity

Sound effects are short duration (usually less than a minute) sounds played by a game to signify action; that is, played when events occur in the game world (such as footsteps played when a character walks, or a door slam sound played when a door is closed, etc.). Sound effects are typically small files that are recorded to disk from microphones or downloaded from sound banks on DVD or online, and they are also typically encoded in WAV (wave) format and loaded entirely in memory by the video games that play them (unlike music, which is streamed rather than loaded; this is explained later). Developers record and edit sound effects for their games using sound effects software. Cross-platform game developers specifically prefer cross-platform sound effects editors so the SFX created on one platform (say Windows) can be opened and edited by using the same editor on another platform (say Linux). It is perhaps not least because of the needs of cross-platform game developers that the cross-platform sound effects editor Audacity has gained recent popularity. Freely available and supporting Windows, Linux, and Mac, Audacity is claimed to be the 11th “most popular” download from SourceForge.net with over 24 million downloads, and was also the winner of the SourceForge.net Community Choice Award for Best Project for Multimedia. Audacity can record audio from microphones, edit audio files, and encode audio into a number of popular file formats (such as WAV, OGG, and MP3). Specifically, Audacity boasts the following features:

- Record from microphone, line input, or other sources; 16-bit, 24-bit, and 32-bit (floating-point) samples
- Record up to 16 channels at once (requires multi-channel hardware)
- Edit sounds using cut, copy, paste, and delete
- Import and export WAV, AIFF, AU, and Ogg Vorbis files

6.2.1 Downloading and Installing Audacity on Linux Ubuntu

1. Beginning from the Ubuntu desktop, select **System | Administration | Synaptic Package Manager**.

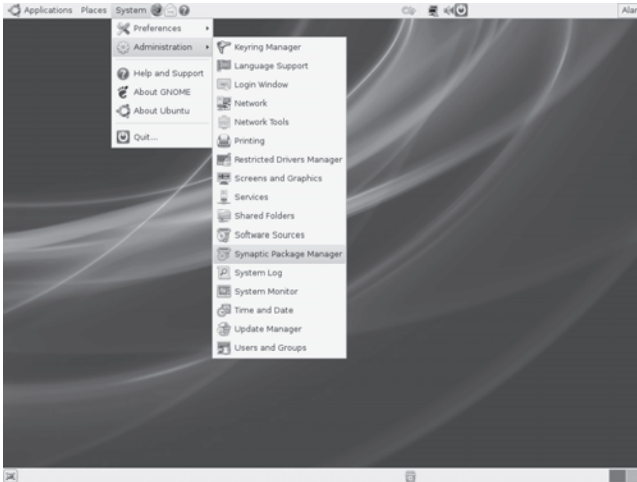


Figure 6-2

2. Search the application repository for Audacity and mark this application using the check box in the application list view where Audacity appears as an option for installation. Click the **Apply** button to install it to the system.

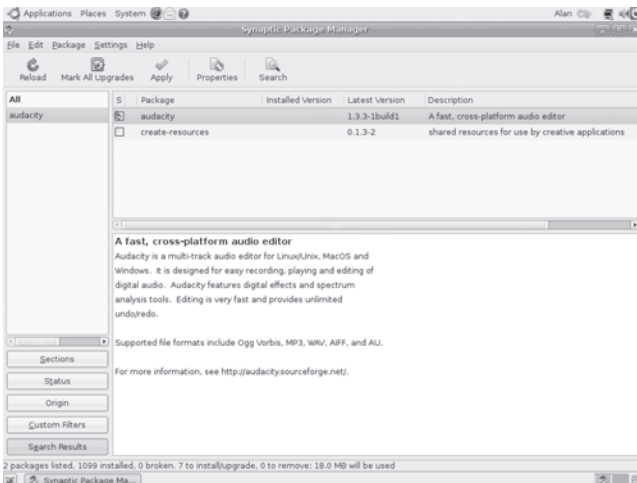


Figure 6-3

- Once Audacity is installed to the system, it can be launched from the Ubuntu main menu via **Applications | Sound & Video | Audacity Sound Editor**.

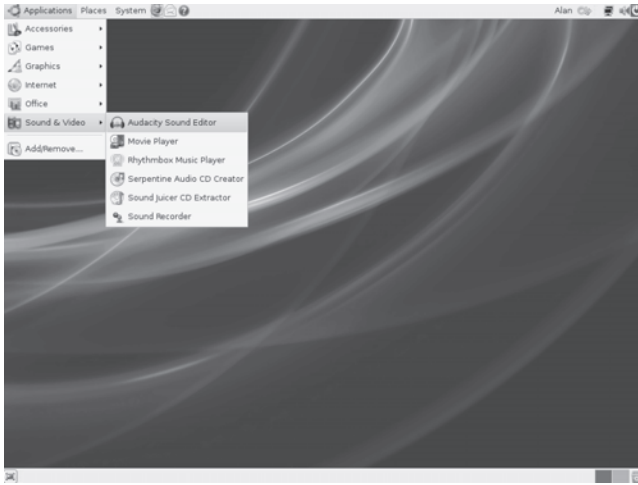


Figure 6-4

Audacity is now ready to use.

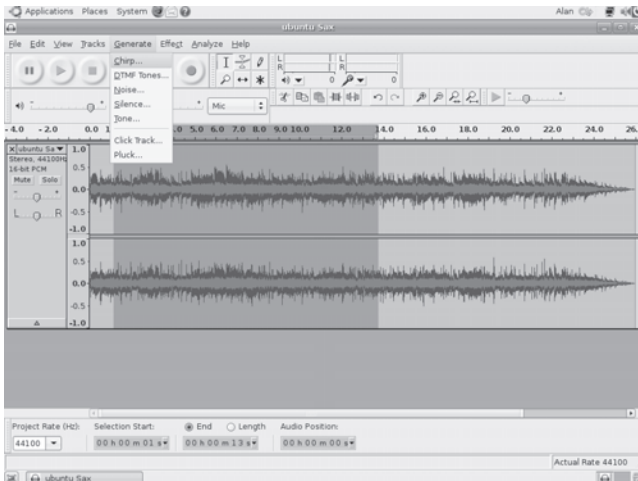


Figure 6-5

6.2.2 Downloading and Installing Audacity on Windows or Mac

1. Beginning from the desktop, navigate a web browser to the Audacity web site at <http://audacity.sourceforge.net/>.
2. Click **Download Audacity** from the web page to proceed to the downloads area.

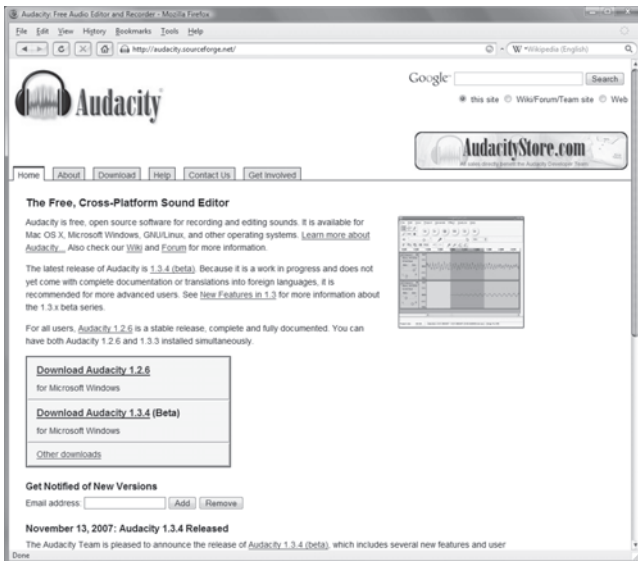


Figure 6-6

3. Download the appropriate installer for your OS; the Windows installer for Windows, Mac OS X for Mac, etc. This download page also hosts Audacity for Linux distributions other than Ubuntu.

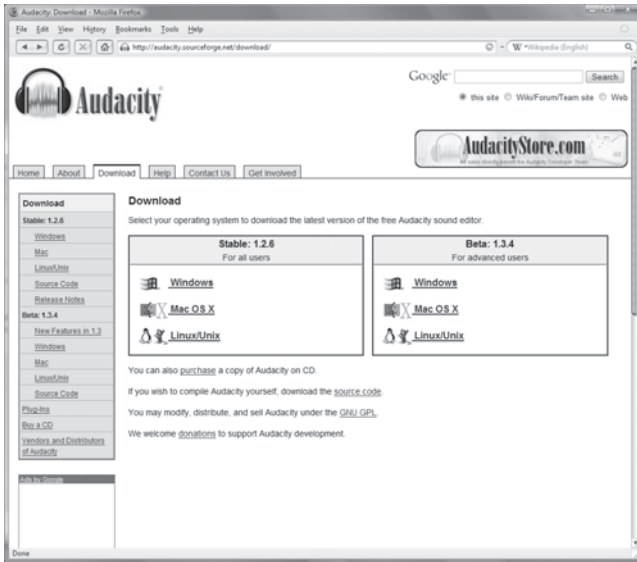


Figure 6-7

4. Follow the downloaded installation wizard to install Audacity to the system.



Figure 6-8



TIP. This chapter focuses primarily on programming sound for games, and as such does not examine Audacity in depth as an application in terms of recording, editing, and exporting sounds. More information on Audacity (including tutorials for getting started) can be found on the Audacity Wiki documentation web site at <http://www.audacityteam.org/wiki/>.

6.3 Recording/Creating and Editing Music

Unlike sound effects, which have a short duration and are designed to represent action without visuals, game music is usually longer in duration (more than 60 seconds) and is typically designed to be unobtrusive and also “loopable” (that is, sounds the same at the beginning and end of the piece) so that a single track may be played seamlessly in the background for as long as the player remains in the current scene (level, map, room) of the game. Game developers typically acquire music for their games via at least one of the following means:

- **Purchased music** — This includes sound effect compilation CDs or DVDs; music can also be purchased from “music bank” CDs, such as those available from Royalty Free Music at <http://www.royaltyfreemusic.com/>.
 - **Homemade music** — Some game developers with musical talents produce their own music in-house, using either commercially available (and often not cross-platform) music creation software like eJay, Fruity Loops, or Cubase, or by recording their live sessions and then later editing the recorded tracks using sound effect editing software. This chapter will examine a cross-platform and freely available music creation package called Schism Tracker (http://sovietrussia.org/wiki/Schism_Tracker).
 - **Contracted music** — Many (perhaps most) developers (especially independent developers) seek out a band or musician to whom they can outsource their music development. Many bands and their music (most available under the Creative Commons License) can be found at the free music site Jamendo at <http://www.jamendo.com/>.
-

6.4 Music Creation Software

Developers aiming to create their own music for their games may, in the present computing climate of patented codecs and other digital rights management (DRM) “enabled” media, may find it difficult to come across a completely free *and* cross-platform music creation solution. There are a number of commercial options, from Cubase to Fruity Loops, each designed with a polished GUI intended to make the process of electronic music creation simpler for the developer. However, in the world of free and cross-platform software (Windows, Linux, and Mac) there is Schism Tracker for making music.

6.4.1 Downloading and Installing Schism Tracker on Linux Ubuntu

1. Beginning from the Ubuntu desktop, start the Synaptic Package Manager from the main menu by choosing **System | Administration | Synaptic Package Manager**.

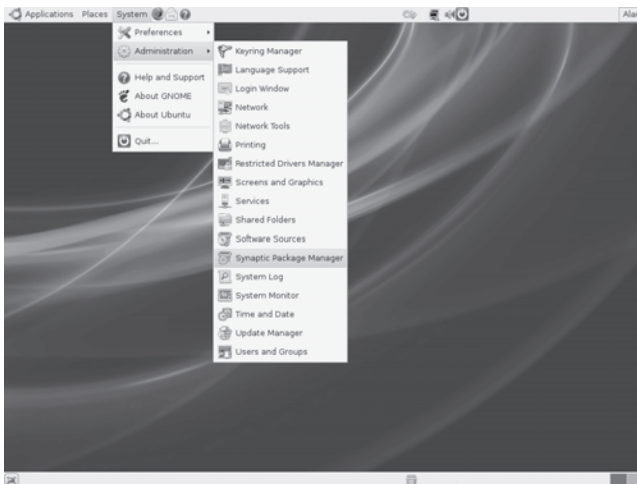


Figure 6-9

- In Synaptic Package Manager, search the Ubuntu application repositories for Schism Tracker, and then mark it for installation by checking the check box in the resulting application list. Click **Apply** and Schism Tracker will be installed.

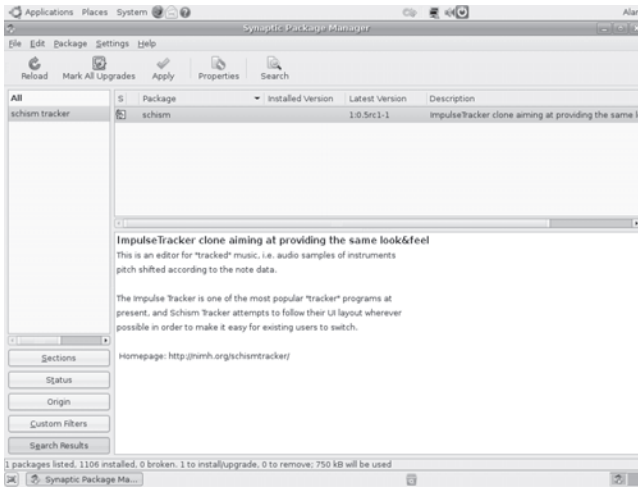


Figure 6-10

- Launch Schism Tracker from the Ubuntu Terminal by clicking **Applications | Terminal** from the Ubuntu main menu, and at the terminal enter:

```
schism tracker
```

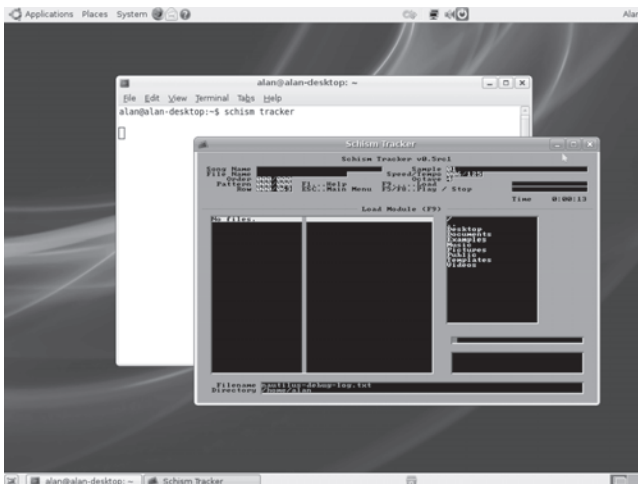


Figure 6-11

6.4.2 Downloading and Installing Schism Tracker on Windows and Mac

1. Beginning from the desktop, navigate a web browser to the Schism Tracker web site at http://sovietrussia.org/wiki/Schism_Tracker.
2. Download the appropriate package for your operating system; there is also a package available for Linux distributions other than Ubuntu.

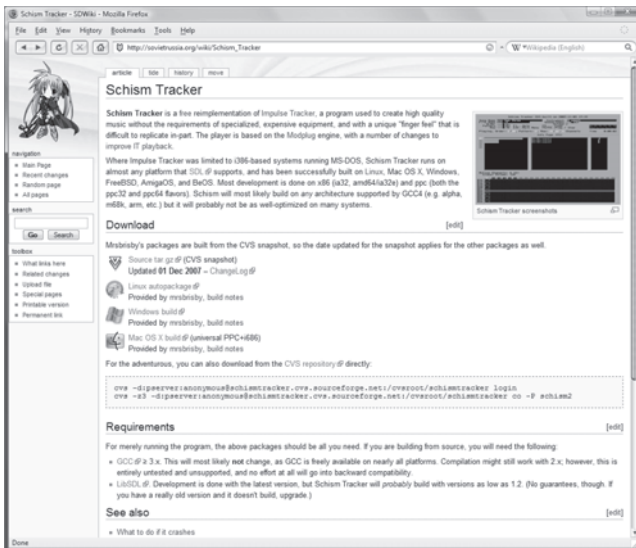


Figure 6-12

3. Extract the compressed zip archive (for Windows) or run the DMG archive for Mac to install Schism Tracker to the system.



Figure 6-13

6.5 Programming Audio with **SDL_mixer**

At its simplest, audio programming for games is about playing sound (footsteps, raindrops) and music (background and incidental) at appropriate moments during game execution. Music is played to set a scene's mood, while sounds are played in response to user actions in the game world, like opening doors or smashing windows. In this section we'll look at a cross-platform API SDL that is free for both commercial and non-commercial usage.

The SDL (Simple DirectMedia Layer) was considered in the previous chapter primarily as a cross-platform graphics API, an API designed to load images from files on disk to surfaces in memory, and then to present the surface pixels frame by frame to the game window at run time. By default, SDL can load pixels only from images in the standard Windows BMP (bitmap) format, unless a plug-in module such as SDL Image is downloaded and installed. This plug-in offers to developers the `Image_Load` function, which extends SDL's range of accepted image file formats to PNG, TGA, JPEG, and several others. In keeping with this notion of modular plug-ins, where additional libraries are downloaded from the Internet to extend SDL's default functionality, the SDL can also play and mix audio files (such as MP3s and OGGs) by way of a downloadable plug-in library called `SDL_mixer`. This plug-in offers to developers a set of functions and data types for playing audio in SDL applications. `SDL_mixer` loads sounds from files on disk to buffers or streams in memory, and from these it can play and mix as many sounds simultaneously as the system's sound hardware supports. `SDL_mixer` supports sounds in the following file formats:

- WAVE/RIFF (.wav)
 - AIFF (.aiff)
 - VOC (.voc)
 - MOD (.mod, .xm, .s3m, .669, .it, .med, and more)
 - MIDI (.mid; using timidity or native midi hardware)
 - Ogg Vorbis (.ogg)
 - MP3 (.mp3)
-

The following step-by-step processes illustrate how `SDL_mixer` can be downloaded, installed, and configured “ready for use” with the Code::Blocks IDE on the Linux (Ubuntu) and Windows platforms.



TIP. The process of installing both Code::Blocks and SDL was explained in earlier chapters of this book.

6.5.1 Installing and Configuring `SDL_mixer` on Linux Ubuntu

1. Beginning from the Ubuntu desktop, start the Synaptic Package Manager from the Ubuntu main menu by selecting **System | Administration | Synaptic Package Manager**.

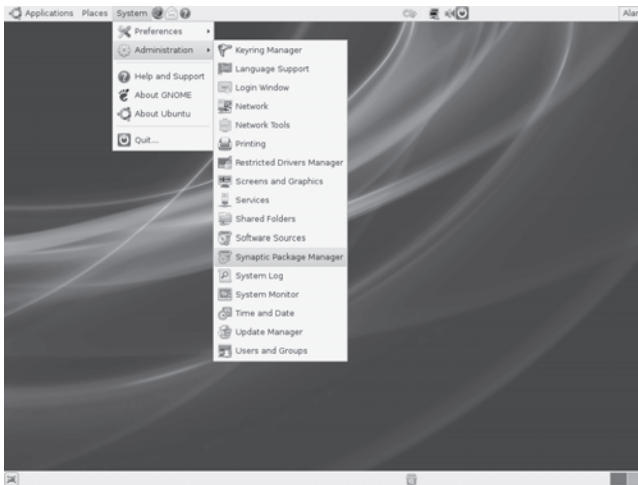


Figure 6-14

2. Search for `SDL Mixer` and click the **OK** button to return in the list view a list of matching applications available to install from the Ubuntu Repositories. Select both the `SDL Mixer` *and* the `SDL Mixer` development files, then click the **Apply** button to install them to the system.

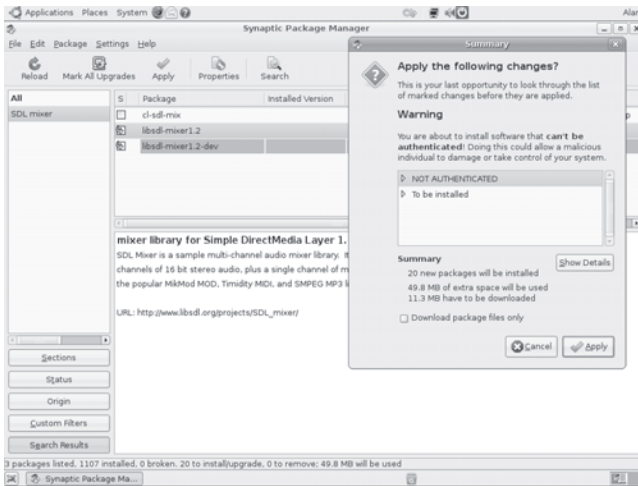


Figure 6-15

3. Close the Synaptic Package Manager and start Code::Blocks from the Ubuntu main menu by selecting **Applications | Programming | Code::Blocks IDE**.
4. Start a new SDL Code::Blocks project or open an existing project. To configure SDL Code::Blocks projects for use with the SDL_mixer libraries, select **Project | Build Options** to view the active project's compilation settings. Click the **Linker settings** tab and add SDL Mixer to the list of libraries linked to by the project's compiler. Also add the following preprocessor directive to the main source file to include the SDL_mixer header file and its types into the project:

```
#include <SDL_Mixer.h>
```

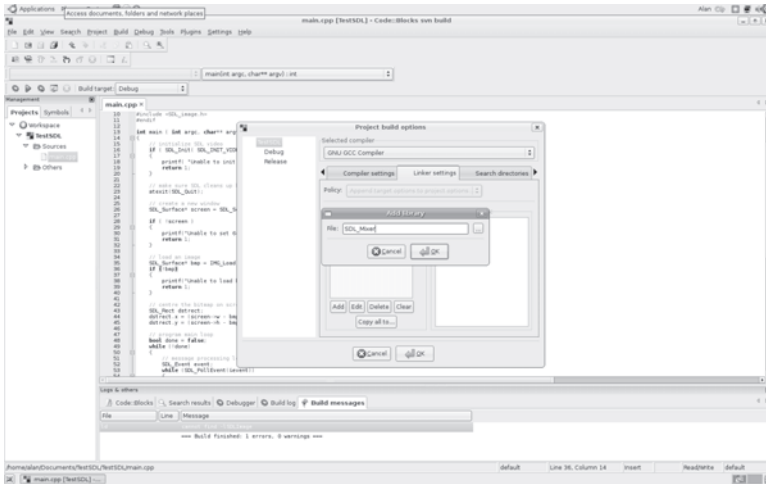


Figure 6-16

6.5.2 Installing and Configuring SDL_mixer on Windows

1. Beginning from the Windows desktop, navigate a web browser to the SDL_mixer web site at http://www.libsdl.org/projects/SDL_mixer/ or visit the SDL home page at <http://www.libsdl.org/>, and browse the searchable database of SDL add-ons for the SDL_mixer add-on.
2. At the SDL_mixer web site, download both the Win32 binaries and the Win32 development libraries to the desktop, and from there extract the contents of the downloaded ZIP archives into the existing SDL development folder on the hard disk (the folder containing the SDL header files and lib files).



Figure 6-17

3. Start Code::Blocks and either create a new SDL project using the Code::Blocks SDL Wizard or load a previously saved project. Then select **Project | Build Options** from the Code::Blocks main menu to view the active project's compilation settings. Click the **Linker settings** tab and add `SDL_mixer` to the list of libraries linked to by the project's compiler. Also add the following preprocessor directive to the main source file to include the `SDL_mixer` header file and its types into the project:

```
#include <SDL_Mixer.h>
```

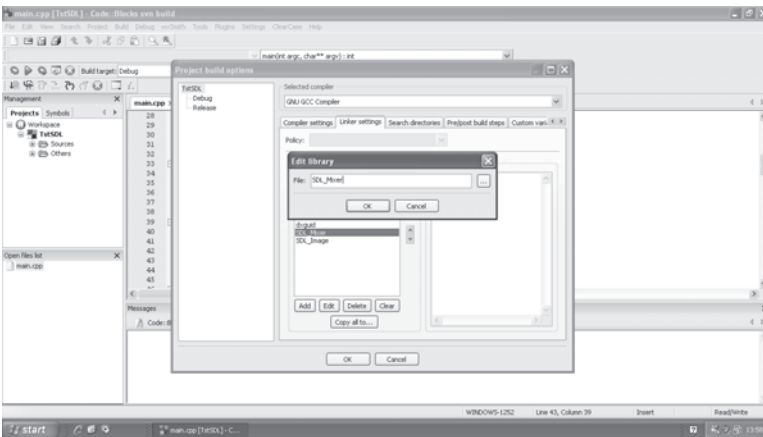


Figure 6-18



NOTE. The `SDL_mixer` packages (binaries and development libraries) downloaded from the `SDL_mixer` web site together include an aggregation of run-time DLLs, specifically `SDL_Mixer.dll`, `smpeg.dll`, `libogg-0.dll`, `libvorbis-0.dll`, and `libvorbisfile-3.dll`. All of these DLLs must be distributed to end users alongside the compiled `SDL_mixer` application if the application is to run successfully; that is, every DLL should be included in the same folder from which the compiled executable is run on the end user's system.

6.5.3 Initializing the `SDL_mixer` Library

The `SDL_mixer` library is an add-on for `SDL` (though it can be used independently of the `SDL` itself) and it offers to game developers a set of functions for playing and mixing both music and sound effects in their games. Programmatically, the first step in initializing the `SDL_mixer` library should be a function call to `Mix_OpenAudio`, and

this function should be called before any other in the `SDL_mixer` library. The `Mix_OpenAudio` function takes the following form:

```
int Mix_OpenAudio(int frequency, Uint16 format, int channels,
                  int chunksize)
```

- **int frequency** — Output sampling frequency measured in Hz of sounds played via the speakers. For many games this value will be 22050 (`MIX_DEFAULT_FREQUENCY`).
- **Uint16 format** — Specifies the format of the audio to be processed by the hardware (in terms of bits per sample). This value will typically be `MIX_DEFAULT_FORMAT`, which equates to `AUDIO_S16SYS`. The possible values for this argument are as follows:
 - **AUDIO_U8** — Unsigned 8-bit samples
 - **AUDIO_S8** — Signed 8-bit samples
 - **AUDIO_U16LSB** — Unsigned 16-bit samples, in little-endian byte order
 - **AUDIO_S16LSB** — Signed 16-bit samples, in little-endian byte order
 - **AUDIO_U16MSB** — Unsigned 16-bit samples, in big-endian byte order
 - **AUDIO_S16MSB** — Signed 16-bit samples, in big-endian byte order
 - **AUDIO_U16** — same as `AUDIO_U16LSB` (for backward compatibility)
 - **AUDIO_S16** — same as `AUDIO_S16LSB` (for backward compatibility)
 - **AUDIO_U16SYS** — Unsigned 16-bit samples, in system byte order
 - **AUDIO_S16SYS** — Signed 16-bit samples, in system byte order
- **int channels** — The number of channels that may be used by the API for playing sounds; refers typically to the number of speakers for which a game's audio is designed: 1 for mono, 2 for stereo, etc.
- **int chunksize** — Bytes per output sample. This could be 4096.

Sample code:

```
if(Mix_OpenAudio(22050, MIX_DEFAULT_FORMAT, 2, 4096) == -1)
{
    return 1;
}
```

6.6 Sounds and Music with `SDL_mixer`

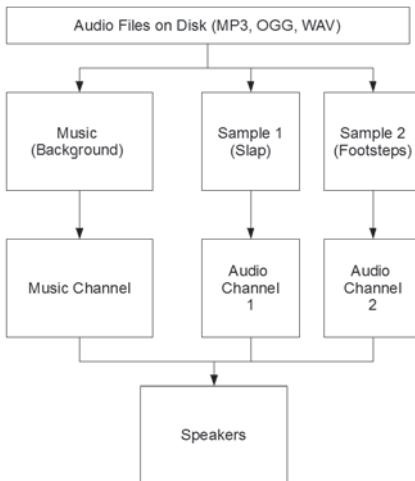


Figure 6-19

The `SDL_mixer` add-on divides audio into two types: samples or music (sound or music). The term “sample” refers to a sound of short duration (less than a minute in length) and includes effects such as footsteps, gunshots, screams, and potentially millions of others that may occur in a game. By contrast, “music” refers to a longer, instrumental soundtrack designed to play subliminally or incidentally (in the background), setting the mood at any one time for any given scene in the game. Thus, audio can be loaded from files on disk and into `SDL_mixer` as either a sample or as music, reflecting how a developer intends the audio to be played in the game.

Audio loaded into `SDL_mixer` as a sample (and not as music) is loaded in its entirety from its file, byte by byte, into a buffer in memory where it can later be played on demand. It can even be played simultaneously with other samples also playing via the sound hardware's audio channels. In other words, `SDL_mixer` can play one or more samples simultaneously. On the other hand, `SDL_mixer` may play only one song (music) at any one time. Furthermore, since music is typically longer in duration than a sample and thereby consumes more memory on disk, `SDL_mixer` does not load a song entirely into memory like it does a sample; rather, a song is said to be streamed from a file and into memory byte by byte during playback. Thus, for `SDL_mixer`, music playback is the process of intelligently loading from a file only those segments of a song currently relevant to immediate playback while simultaneously discarding from memory all of those previously loaded segments that have since become irrelevant for playback. In this way, chunks of the song are loaded and unloaded on-the-fly as the song is being played, such that a song is never entirely present in memory but never wholly out of memory. Both the sample and music data types of `SDL_mixer` are now considered in further detail.

6.6.1 Loading Music

Programmatically, music is first loaded from a file on disk (such as MP3 or OGG) and into a memory buffer (not in its entirety) by the `Mix_LoadMUS` function in preparation for playback later. This function accepts as an argument a valid file name to a music file on disk, and it returns a `Mix_Music` handle; that is, it returns a pointer to a memory buffer containing the music partially loaded from the file. The `Mix_LoadMUS` function takes the following form:

```
Mix_Music *Mix_LoadMUS(const char *file)
```

Sample code:

```
music = Mix_LoadMUS( "Music.ogg" );
```



NOTE. Valid file formats are WAV, MOD, MIDI, OGG, and MP3.

6.6.2 Playing Music

Using `SDL_mixer`, music is loaded from a file on disk and into a `Mix_Music` memory buffer by the `Mix_LoadMUS` function in preparation for playback later. `SDL_mixer` offers to developers at least three different functions from which to choose to begin playback of a memory buffer, each function beginning playback of a song in a different way:

- **Mix_PlayMusic** — This function begins playback of a song at full volume from the beginning of the song.
- **Mix_FadeInMusic** — This function likewise begins playback of a song from the beginning, but the song begins mute and its volume gradually increases to full volume over a specified fade-in time measured in milliseconds.
- **Mix_FadeInMusicPos** — This function begins playback of a song from a point other than at the beginning (from a specified time offset as measured in milliseconds from the beginning of the song) and playback also begins at mute volume, graduating to full volume over a specified fade-in time. Consider the following code.



NOTE. Remember, there are 1000 milliseconds in 1 second. So 2000ms = 2s; and 7500ms = 7.5 seconds.

Standard Play Music Function

```
// Mix_Music *music;
// Already loaded

//Prototype is: int Mix_PlayMusic(Mix_Music *music, int loops)
//Loop = Number of times to repeat sound playback; where -1 is
infinite
if(Mix_PlayMusic(music, -1)==-1)
{
    //Error occurred here
}
```

Play Music Using the `SDL_mixer FadeIn` Function

```
// Mix_Music *music;
// Already loaded

//Prototype is: int Mix_FadeInMusic(Mix_Music *music, int loops,
int ms)
//Loop = Number of times to repeat sound playback; where -1 is
infinite
//ms = milliseconds during which music fades in from mute to full
volume
if(Mix_FadeInMusic(music, -1, 2000)==-1)
{
    //Error occurred here
}
```

Play Music from a Specified Time Offset Using the `FadeIn` Function

```
// Mix_Music *music;
// Already loaded

//Prototype is:
//int Mix_FadeInMusicPos(Mix_Music *music, int loops, int ms,
double position)
//Loop = Number of times to repeat sound playback; where -1 is
infinite
//ms = milliseconds during which music fades in from mute to full
volume
//position = Offset in milliseconds from the beginning of the
//song where playback is to begin
if(Mix_FadeInMusicPos(music, -1, 2000)==-1)
{
    //Error occurred here
}
```

6.6.3 Controlling Music

In addition to streaming music using the three playback functions, `SDL_mixer` also offers to developers functions for controlling or stopping music playback in the same way a media player or stereo system offers buttons to start, stop, rewind, pause, and forward the contents of a media stream. These media control functions include the following:

- **int `Mix_VolumeMusic`(int volume)** — Sets the volume of the music currently being played by `SDL_mixer` according to the integer argument (*Volume*). This argument can be any integer from 0 (mute) to 128 (full volume).

Example:

```
//Sets the volume
Mix_VolumeMusic(128);
```

- **void `Mix_PauseMusic`()** and **void `Mix_ResumeMusic`()** — Pauses and resumes playback of music, respectively.

Example:

```
//Pause
Mix_PauseMusic();
```

```
//Resume
Mix_ResumeMusic();
```

- **int `Mix_SetMusicPosition`(double position)** — Seeks to a specified playback position in the music currently being played by `SDL_mixer`, specified by the argument (*position*) as an offset in milliseconds from the beginning of the song.

Example:

```
//Plays from specific point
Mix_SetMusicPosition(50);
```

- **int `Mix_HaltMusic`()** — Stops playback of the song currently being played by `SDL_mixer`.

Example:

```
//Stops music playback
Mix_HaltMusic();
```

- **int `Mix_FadeOutMusic`(int ms)** — First fades out and then halts the music currently being played by `SDL_mixer`.

Example:

```
//Stops music playback
Mix_FadeOutMusic(5000);
```

- void **Mix_FreeMusic**(Mix_Music *music) — Deletes a song held in memory. This function should be called after a song is no longer needed for further playback using `SDL_mixer`.

Example:

```
//Frees memory buffer  
Mix_FreeMusic(Music);
```

6.6.4 Playing Samples through Channels in `SDL_mixer`

In `SDL_mixer`, a “sample” refers to a sound that is typically less than a minute in duration. The sample is loaded from a file on disk (.wav, .ogg, etc.) and into a memory buffer in the system memory where it waits in situ to be played on demand for as many times as required. The sample held in system memory may, for example, be the sound of footsteps intended to be played while an NPC walks around the game world or perhaps the sound of a gunshot that is to occur whenever a weapon is fired. In every case, however, a sample is held in memory by `SDL_mixer`, and it must be sent through at least one of the available audio channels on the sound hardware if it is to be heard by the player through the system’s speakers. Each sample travels along its channel at the same speed as any other sample on any other track, and each audio channel accommodates only one sample at any one time, meaning that different samples must travel along different channels if they are to be heard simultaneously on the speakers by the player. Thus, the total number of samples that may play simultaneously on the speakers corresponds to the total number of audio channels supported by the system’s sound hardware.

The following sections describe how to program with samples and channels using `SDL_mixer`, including how to load sounds from files on disk and into memory buffers as samples; how to assign a sample to an audio channel for playback; how to control playback of individual samples playing in a channel through positioning, rewinding, and pausing; and how to delete samples from memory when further playback of a sample is no longer required.

6.6.5 Loading Sounds into SDL_mixer as Samples

Sounds are first loaded from files on disk (.wav, .ogg, etc.) into SDL_mixer as samples in memory using the Mix_LoadWAV function, and are then ready for later playback via audio channels. This function accepts as a string argument a valid file name of a sound file on disk to load as a sample. Mix_LoadWAV takes the following form:

```
Mix_Chunk *Mix_LoadWAV(char *file)
```

Example code:

```
// load sample.wav in to sample

Mix_Chunk *sample;
sample=Mix_LoadWAV("sample.wav");
if(!sample)
{
    //Error
}
```



NOTE. Every sample loaded into SDL_mixer using Mix_LoadWAV must later be destroyed (freed from memory) with a call to the Mix_FreeChunk function. This function must be called once per loaded sample after it is no longer required for further playback in the game.

6.6.6 Handling Channels with SDL_mixer

Programmatically, sounds are loaded from files into SDL_mixer as samples, and samples are played through audio channels; only one sound may play in any audio channel at any one time. Thus, the total number of samples that may play simultaneously using SDL_mixer is necessarily limited to the number of audio channels supported by the sound hardware on the player's system; 16 channels support up to 16 simultaneous sounds, 8 channels = 8 sounds, etc. Developers can manually set the total number of audio channels to be used by SDL_mixer for any given game using the Mix_AllocateChannels function. This function takes the following form and may be called

anywhere in an `SDL_mixer` application (even when sound is currently playing through audio channels):

```
int Mix_AllocateChannels(int numchans)
```

Example code:

```
// allocate 16 mixing channels
Mix_AllocateChannels(16);
```

`SDL_mixer` offers a variety of functions from which to choose for playing samples in channels, and like their counterpart music playing functions, each function differs in the way the samples are played. The following sample playing functions are available:

- **Mix_PlayChannel** — This function plays a specified sample (for example, a punch sound or a gunshot) in a specified channel immediately at full volume and from the beginning of the sample (from time 0).

```
int Mix_PlayChannel(int channel, Mix_Chunk *chunk, int loops)
```

Example code:

```
// channel = play sample on first free unreserved channel (-1)
// loops = play it once only
// Mix_Chunk *sample; //previously loaded
if(Mix_PlayChannel(-1, sample, 0)==-1)
{
    //Error occurred here
}
```

- **Mix_PlayChannelTimed** — This function plays a specified sample in a specified channel immediately at full volume and from the beginning of the sample, playing for a specified period of time after which playback stops.

```
int Mix_PlayChannelTimed(int channel, Mix_Chunk *chunk, int
    loops, int ticks)
```

Example code:

```
// play sample on first free unreserved channel
// play it for half a second
// Mix_Chunk *sample; //previously loaded
if(Mix_PlayChannelTimed(-1, sample, -1 , 500)==-1)
{
    //Error occurred here
}
```



NOTE. Other channel playing functions include:

- void Mix_Pause(int channel)
- void Mix_Resume(int channel)
- int Mix_HaltChannel(int channel)
- int Mix_FadeOutChannel(int channel, int ms)

6.7 Conclusion

This chapter considered the fundamentals of the `SDL_mixer` library, an add-on for the `SDL` API that is designed for playing game audio in the form of samples (sounds) and music (background or incidental). Both sounds and samples are loaded one by one from files on disk (such as `.mp3s`) and into memory buffers managed by the `SDL_mixer` library. Samples are loaded from disk in their entirety and are played to the speakers through audio channels, one sample per channel; as many samples may play simultaneously as there are available channels. In contrast to samples, music, which is more memory intensive, is *streamed* from a file (loaded and unloaded on-the-fly, region by region) according to the region or part of the song being played at any one time. This region is played to the speakers through one audio channel only, thus only one song may play at a time. Playback of a new song will stop and replace the playback of any prior song currently being played. In Chapter 11, other audio APIs (namely `FMOD` and `BASS`) will be considered briefly for those who wish to use cross-platform alternatives to `SDL_mixer`.

Chapter 7

Game Mechanics

This book so far has explained how to configure a cross-platform environment on a single machine through dual-booting, across a selection of machines each running a different OS, or running subordinate guest OSs on a single host through virtualization. We've also described a cross-platform C++ IDE called Code::Blocks, which can be used to create and compile C++ cross-platform games. In addition, we've covered programming with two freely available, open-source and cross-platform gaming libraries: SDL (Simple DirectMedia Layer) and SDL_mixer. SDL is a library designed to draw fast-paced graphics and animations to the display in real time, and SDL_mixer is a library for streaming and playing audio from files such as WAV and OGG.

However, this is not all there is to computer games. Games are to some extent holistic creations insofar as they are greater than the sum of their parts, something above and beyond the IDE and gaming libraries (like SDL) from which they are made. Games cannot merely be reduced to essential core components alone, such as graphics, sound, input, etc.; they are an imaginative synthesis of core components working in unison that bring together the graphics, sound, physics, artificial intelligence, story, and genre into a coherent totality. Overall, the specific ingredients for a game (the libraries, IDEs, etc.) may in no way differ from those used for any other game, but it is the particular configuration (the quirky recipe) of those ingredients into a whole that makes each game unique, at least in theory. This chapter is about recipes; it is about making things work together, and this makes the difference between a game and a senseless collection of graphics and sound.



NOTE. Given the broad scope of subjects included here, this chapter is arranged in a Q&A format to make it simpler to skim so readers can jump to specific topics of interest.

7.1 Getting Started with Game Worlds

Q. I want to make a game. I have learned how to program with C++ using Code::Blocks. I know how to draw images to the window with SDL, and I know how to play sound with OpenAL. Let's say I now want to make a 2D RPG game with a top-down view (the camera is looking downward, directly at the player and the rest of the game world). How do I start developing this game?

A. The first stage could be to create a game world, or a map. The *map* is a single graph-like coordinate space in which all game objects — from the player to NPCs and walls and chairs, etc. — exist as physical bodies, each with a specified X and Y position in the map measured from the origin, and each with a specified width and height. This means each game object has a measurable, definable position in the map, and each object can be measured either from the origin of the coordinate space or relative to any other object. This then introduces the notion of a game object as a *base class*; that is, as a set of properties that all objects in the game — whether a player or NPC, moveable or not — share, and it is from this base class that all game objects are *derived* since all objects in the map will have a position and a size. Objects will also have a unique *identifier*, a name or number to single them out from other objects by human readable tags. The following code is a sample base class; the code is followed by a figure that illustrates the geometry.

```
#include <iostream>

using namespace std;

//Vector class for storing (X,Y) position
class cVector
{
private:
protected:
```

```
public:

    long x;
    long y;
};

//Base game object
class CGameObject
{
private:
protected:
public:
cVector m_Position;
long m_Width;
long m_Height;
std::string m_Name;
bool m_Visible;

    CGameObject(std::string Name)

    {
        m_Name = Name;
        m_Position.x = m_Position.y = m_Width = m_Height = 0;
        m_Visible = true;
    }
};
```

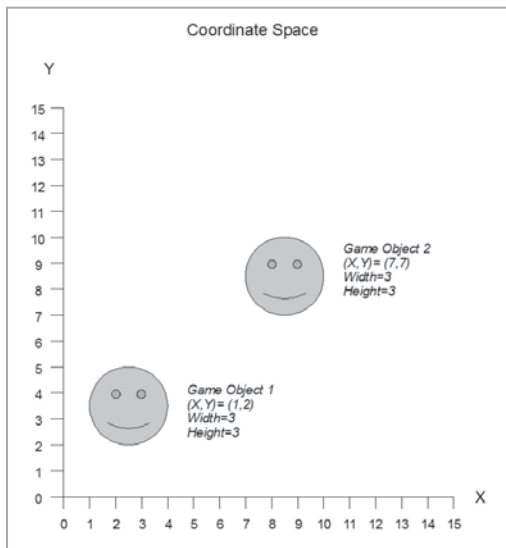


Figure 7-1: The game world map and object positions

7.2 Creating Derivative Objects

Q. Okay; I have created a base object as above, with position, width, height, and name properties. But that in itself is quite basic and hardly useful; it only represents a position and a size. How can I add a player character to the level, for example? Or more generally, how can I add obstacles and other objects to the map that are beyond what the base class offers?

A. The base class is only the foundation class, or the starting point from which all other classes — each of them a game object — will begin. It is not a class intended to be instantiated itself, but represents only the minimum set of properties and methods intrinsic to all game objects; that is, the properties that all objects will inherit. So while every type of game object may differ widely from the others in their implementation (some objects are walls, some are floors, some are NPCs, etc.), the base class contains some properties common to all objects such that, were those properties removed from those objects, they would no longer be game objects. In short, then, every type of game object in a single map is implemented as a class, but one that is first and foremost derived from class “game object.” The following code is a sample derived class, and this is followed by a diagrammed model of derived game objects.

```
class cPlayer : public CGameObject
{
    private:
    protected:
    public:

    long m_Health;
    cVector m_FacingDirection;
    long m_Speed;

    cPlayer(std::string Name) : CGameObject(Name)
    {
        Speed = 0;
        Health=100;
    }
};
```



Figure 7-2: Derived game object hierarchy

7.3 Maintaining Game Objects

Q. I see; so a level may contain potentially hundreds of game objects, from walls and floors to power-ups, tables, chairs, enemies, the player, and too many others to list. This could grow rather unwieldy and difficult to manage, with all these objects and pointers to objects laying around in memory here and there. How do I maintain all of them?

A. Perhaps the simplest solution here would be to maintain a linked list (a `std::vector`) of game object pointers in memory. The `std::vector` class is part of the STL (Standard Template Library) and ships with `Code::Blocks`. STL offers a selection of classes for handling memory and pointers, and `std::vector` is specifically designed for managing lists of pointers, like a dynamic array that can grow and shrink in size exactly to accommodate the right number of pointers. The following sample code illustrates how to keep a list of game objects and includes methods for adding items, deleting items, and clearing all items from the list.

```
#include <vector>

std::vector<CGameObject*> m_GameObjects;

void AddObjectToList(CGameObject *Obj)
```

```
{
    if(Obj)
    {
        m_GameObjects.push_back(Obj);
    }
}

void ClearList()
{
    for(unsigned int i=0; i < m_GameObjects.size(); i++)
    {
        if(m_GameObjects[i])
            delete m_GameObjects[i];
    }

    m_GameObjects.clear();
}

void DelItem(std::string Name)
{
    for(unsigned int i=0; i < m_GameObjects.size(); i++)
    {
        if(!m_GameObjects[i])
            continue;

        if(m_GameObjects[i]->m_Name==Name)
        {
            delete m_GameObjects[i];
            m_GameObjects.erase(m_GameObjects.begin()+i);
            return;
        }
    }
}
```

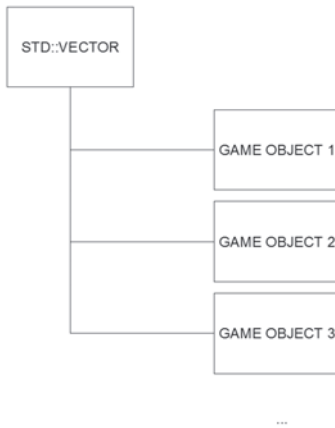


Figure 7-3: A linked list of game object pointers

7.4 Tile-based Levels

Q. Okay; so every object in a game map is derived as a SuperClass from base class `CGameObject`. This means each object in a map — such as the player, enemies, walls, and doors — have at least an X,Y position, a name, and a size. A map, then, is merely a structural arrangement (a collection) of game objects; a place where game objects together form walls, houses, towns, worlds, and other structures in which game characters live and act. So now the question arises: I want to start creating my top-down RPG game. What software should I use to actually build a map to define the X,Y location of every object in the map and position the doors and walls and windows and power-ups? Most games seem to have level editors where developers visually design a level layout, but I have no such editor available. Does this mean I have to hard-code the map? I hope not. Having to hard-code a map line by line in C++ using the `Code::Blocks` editor would not only be tedious and time-consuming since every object must be declared (at least every object's width, height, position, and name), but it would also mean that I'd have to recompile the `Code::Blocks` project every time I made a change to a map. This is just not acceptable.

A. You are correct; many developers use level editors to build maps for their games. Although there is no level editor available for your new game unless you have made one already, the map *does not* have to be

hard-coded. For top-down games like the original *Zelda*, *Dink Smallwood*, or *Dr. Lunatic Supreme with Cheese*, a map can be thought of as a collection of game objects, or more accurately a collection of *tiles*, which are small, repeatable images (like a patch of grass, a pattern of bricks, or a tree). So a map then may be conceptualized more specifically as a cross-section or grid (or two-dimensional array) of equally sized tiles; each tile ordered side by side in columns and rows to form a complete map. At its most fundamental level, this grid-like map arrangement is typically implemented in games by developers as follows:

1. Before creating a map for any tile-based game, an artist will first create a *tile set*, or a palette of tiles. This is a single image file (PNG, BMP, etc.) that features a copy of every unique tile in a single map, together arranged in columns and rows, one tile beside the next. Its purpose is to group together all related tiles into a single bitmap (palette) rather than to keep each tile in a different file.
 2. Programmatically, a tile set image is an index file of tiles. Since all tiles in the file are the same width and height in pixels, and since the tiles are arranged side by side in columns and rows, each tile may then be identified individually by its index in the grid. That is, a tile is numbered by its sequential position in the file, as read from left to right, line by line and tile by tile.
 3. It follows, then, that a map (being a collection of tiles forming a level) has a relationship to the palette or tile set. First, any single map is composed only from tiles featured in a palette, and not from objects loaded from elsewhere. Second, a map is similar to the palette insofar as it is a grid of tiles, but the map may feature any number of tiles, and any number of copies of tiles, all in various configurations to create a meaningful map. For example, wall tiles are combined to make walls, and these are juxtaposed with door and window tiles to form houses, etc.). In short, then, the map is not a palette to contain the pixels of the tiles themselves because it may duplicate many tiles (housing tiles is reserved for the palette since it features only one copy of each tile). Instead, the map is merely a matrix of numbers in memory, an array of references into the palette that defines only the arrangement (or position)
-

each copy of a tile from the palette should take in the map. In this sense, the map is a finite state machine (a snapshot of a moment) because the numeric value of each element in the map array specifies which tile is occupying that space at any one time. This means that maps need not be hard-coded. Why? Because they are a grid of numbers, and so they could just as easily be loaded from a text file or an XML file, and they could also be output to such files from a custom-made map editor, and even copied to the clipboard in numerical form.

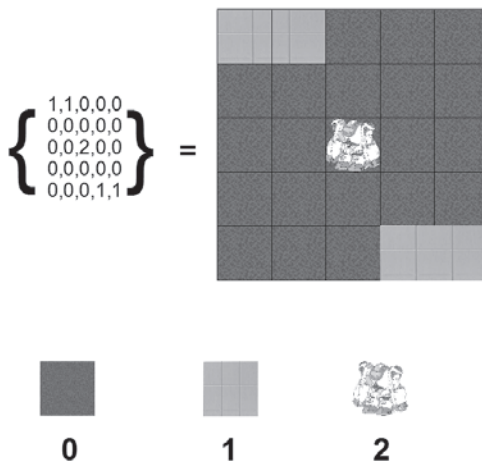


Figure 7-4: A tile-based map

The following example loads a sample tile set from an image file, then arranges the tiles in a map, and finally draws the map to the display using the SDL graphics library.

```

void createLevels()
{
    //Load SDL tiles
    g_LevelTiles = SDL_LoadBMP("Tiles.bmp");

    //Level 1
    //Example of hard-coded level, but most would be defined in a
    //text file

    cLevel* Level1 = new cLevel();

```



```

Level1->m_LevelNumber = 0;

Level1->m_Level[0][2] = OBJECT_WALL; //1
Level1->m_Level[0][3] = OBJECT_WALL; //1
Level1->m_Level[0][4] = OBJECT_WALL; //1
Level1->m_Level[1][2] = OBJECT_DOOR; //0
Level1->m_Level[1][4] = OBJECT_WINDOW; //3
Level1->m_Level[2][2] = OBJECT_FENCE; //4
Level1->m_Level[2][4] = OBJECT_GRASS; //2
Level1->m_Level[2][5] = OBJECT_GRASS; //2
Level1->m_Level[2][6] = OBJECT_GRASS; //2
Level1->m_Level[2][7] = OBJECT_CAR; //5

//More stuff defined[...]

    AddToLevelList(Level1);
}

//-----

//Update level; argument is SDL back buffer; draws map to the
//display
void update(SDL_Surface *Screen)
{
    if((g_LevelTiles)
    {
        for(int indx1 = 0; indx1 < 20; indx1++)
            for(int indx2 = 0; indx2 < 20; indx2++)
            {
                //Get current tile
                GAME_TYPE_OBJECT Index = m_Level1[indx1][indx2];

                SDL_Rect SourceRct;
                SourceRct.x = Index * m_TileSize; SourceRct.w =
                    m_TileSize;
                SourceRct.y = 0; SourceRct.h = m_TileSize;

                SDL_Rect DestRct;
                DestRct.x = indx2 * m_TileSize; DestRct.w = m_TileSize;
                DestRct.y = indx1 * m_TileSize; DestRct.h = m_TileSize;
            }
        }
}

```

```
SDL_BlitSurface(g_LevelTiles, &SourceRct, Screen,
                &DestRct);

//Draw player object

Player->update(Screen);
    }
}
}
```

7.5 Animations and States

Q. So granted, tile-based levels seem an acceptable solution for designing complex maps because each tile can be arranged in a grid-like system of numbers where each cell in the grid references an index/offset in the tile set palette. But there is a further problem. Some tiles in a map — such as the player tile, or an enemy, or a door or window tile — may animate or change state. For example, a door tile can be in one of two different states: open or closed. Or the player tile may potentially enter more states (though only one state at any one time) like a “running” state, or an “attacking” state, or an “idle” state, and in each state the appearance of the player tile will change accordingly. In other words, some tiles do not remain static, but can change through a form of state-based animation. What is the best solution for this?

A. The tile-based map system — a grid of indices referencing tiles in a palette — is itself already half of the solution to the problem of animation. Extending this framework such that every palette includes every frame (tile) of animation (e.g., a tile for the player in each state of walking, running, and jumping), then the concept of animating tiles simply means animating (*changing*) indices in the map grid to refer to different tiles (frames) from the palette at run time.

7.6 Movement

Q. I get the idea; tiles in a map change their appearance based upon their palette index in the map grid. This is because the index of each map cell refers to a unique tile in the palette. I even realize this indexing concept could be extended to create moveable tiles (tiles that can wander around a map, from cell to cell) such as the player tile. For example, let's say the player tile index is 5, and the standard empty grass tile is 4; so a given map will have only one cell set to 5 (since there is only one player character in the map) but many cells can be 4 since the player may be standing in wide-open space, like a field. Now let's suppose the gamer presses the right arrow key and moves the player tile one cell to the right. This has the effect of shifting the number 5 from the current cell to the neighboring cell to the right, leaving behind the old cell now set to 4 (grass) instead of 5 (player). This is all well and good, but moving from cell to cell in this way hardly appears smooth on-screen. To anybody watching, the player's stilted movement as he jumps across the width of one space to the edge of another will make the grid arrangement obvious. In addition, as it stands, the player can only move left, right, up, or down since the grid is formed of equally sized square tiles arranged in columns and rows; so the player cannot move diagonally or at any other angle. In short, I want a smooth-moving, free-roaming player, one that isn't "locked" into the grid.

A. Despite the brilliance and simplicity of a tile-based grid arrangement where maps become nothing more than a grid of numbers, there are undoubtedly limitations concerning tiles that move for the reasons you mentioned, like with the player character. To solve this problem a distinction must then be made that divides tiles into two kinds: static and moveable. Static tiles are walls, floors, and windows. Moveable tiles are the player, NPCs, and cars. Static tiles are locked into the grid, while moveable tiles may be released from the grid and may exist in an unconstrained space, not restricted by a cell's width or height. However, this freedom brings about some consequences worth considering. Firstly, free movement means that it is possible for moveable tiles to exist inside more than one cell in the grid at any one time. This is because, as a tile moves smoothly across a border joining one cell to

another, the tile (as it passes) will be partially in the cell that it enters and partially in the other that it is leaving. Secondly, free movement involves using *vectors* to perform transformations to move the tile from the source point to the destination point. Let's then take a quick look at vectors. More detailed information about vectors and their relevance to game programming can be found in my previous book, *Introduction to Game Programming with C++*.

7.6.1 Movement with Vectors

A sample class for a vector might look like this:

```
class CVector
{
    private:
    protected:
    public:
        long x;
        long y;
};
```

As you can see, a vector is a mathematical construct representing direction. Vector classes are like coordinate and point classes, which feature X and Y ordinate pairs; however, vectors differ from coordinates since they express a *direction*, and not a *location*. Coordinates are used to answer the question “Where?” Vectors are used to answer the question “Which way?” Coordinates specify an absolute position measured from the origin; conversely, a vector may express a direction relative to any point, not just the origin. The vector (1,1) would specify a direction up and to the right at an angle of 45 degrees; and its opposite direction would be (-1,-1).

The vector (8,8) represents the same direction as (1,1), and the same direction as (2,2), but each vector expresses more than simply a direction; it also expresses a distance over which to travel, called a

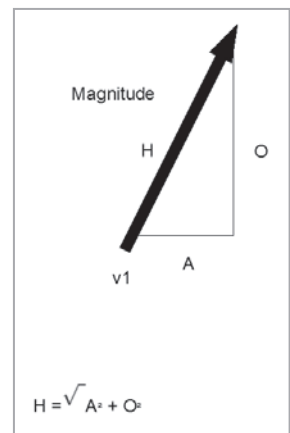


Figure 7-5: A vector's magnitude

vector's magnitude. The *magnitude* is the distance along the diagonal, which is also the length of the hypotenuse.

Vectors can be multiplied by both positive and negative numbers to scale their magnitude and to sometimes change their direction. For example, $(2,2) * 2$ results in a vector of different magnitude but of the same direction $(4,4)$. However, $(2,2) * -3$ results in a vector of different magnitude and of different direction $(-6,-6)$.

Sometimes it is useful for a vector to have a magnitude greater than 1, but often developers will want to express only direction; and this can be expressed by a special vector whose magnitude is 1 called a *normalized* vector, or a *unit* vector. Multiplying a unit vector by any number is like any number multiplied by 1; the result of this multiplication is a vector whose magnitude is the same as the multiplicand. Thus, a vector of any given magnitude (e.g., 5,5) can be stripped of its magnitude and reduced to expressing only its direction by becoming a unit vector through being normalized. A vector is *normalized* by dividing each of its ordinates by its magnitude, as in $(x/mag)(y/mag)$.

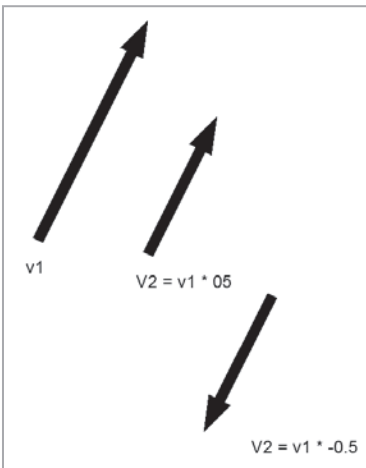


Figure 7-6: Vector scaling

In addition, a vector's direction can be rotated around its origin by a specified angle in degrees. That is, a vector can be rotated around an arc, around the circumference of an invisible circle, by a specified angle. The following code features a complete vector class for the purposes of this chapter.

```
#ifndef VECTOR_H_INCLUDED
#define VECTOR_H_INCLUDED

#include <math.h>

class CVector
{
private:
protected:
public:

float x;
float y;

CVector()
{
x = y = 0.0f;
}

CVector(float px, float py)
{
x = px;
y = py;
}

//Returns vector magnitude
float length()
{
return sqrt((x * x) + (y * y));
}

void normalize()
{
float vecLength = length();

x=x/vecLength;
y=y/vecLength;
}

void scale(float scalar)
{
x=x*scalar;
```

```
        y=y*scalar;
    }

void rotate(float angle)
{
    float veclength = length();

    float tmpx = cos(angle);
    float tmpy = sin(angle);

    x = tmpx * veclength;
    y = tmpy * veclength;
}
};

#endif // VECTOR_H_INCLUDED
```

Using vectors, the following strategy can be followed for creating free moving tiles suitable for the player, NPCs, and others. These tiles may move freely and independently of the map grid, perhaps crossing the border between any two cells, and then stopping there as its area partially intersects one file and partially intersects another. Let's assume a free-roaming player character in a top-down RPG is to be made. Pressing the up arrow walks the character forward in the direction he is currently facing, pressing the down arrow walks the character backward away from the direction he is facing, and the left arrow and right arrow rotate the character counterclockwise or clockwise, respectively, to face a new direction.

Movement for the player (walking forward and backward) is primarily based around its LookAt vector; that is, a player's movement is determined largely by the *direction* in which the player is looking. Hence, moving the player forward or backward means moving the player positively or negatively toward or away from the direction it's facing; while turning left or right refers to the changing, or *rotation*, of the direction vector itself as the player revolves to face new directions. Thus, the player tile (or any free moving tile) as a derived class of CGameObject should maintain a *normalized* direction vector as a property of the class, a vector mathematically expressing the direction in which the player tile is currently facing.

Next, pressing the up arrow or down arrow moves the player forward or backward, respectively, in the direction it's facing. During movement, the player moves from the current position to a destination along the direction it's facing at a speed measured in distance per second (such as 5 pixels per second), meaning any one player may cover the same distance in the same direction during any given interval. This calculation is currently ignoring any environmental weightings of the player's route that may directly or indirectly affect the speed of travel such as terrain type (rocky, snow, slippery, etc.). It is currently assumed that all terrain affects a player's speed equally. Overall, a player tile class (if it is to move freely about a map) must in addition to maintaining a LookAt vector also keep track of an X,Y position on the map and a speed (distance to travel per second). Given these properties, and assuming the player is pressing the up arrow, a tile may calculate the distance and the direction in which it is to move (this process would be the reverse for the down arrow).

Look At Vector

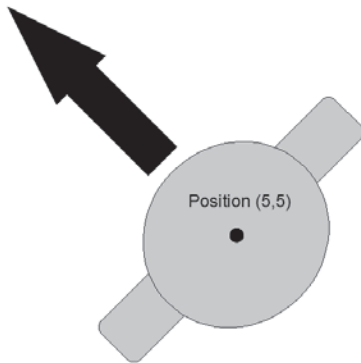


Figure 7-7

On each frame of the game loop, determine if the up arrow on the keyboard is pressed. If true, the player should move forward at a specified speed, from the current position to a new position, in the direction it's facing. The direction in which the player is facing is represented by the LookAt vector, and the distance (length of diagonal) over which it should move during any interval of time is determined by the player's speed per second since $\text{distance} = \text{speed} * \text{time}$. In other words, the player's *normalized* LookAt vector (the direction to move) should be scaled (multiplied) by the distance to travel, and the resultant vector is

the destination to which the player should move, measured as an offset from the player's current position. The code to achieve this follows:

```
//Called once per frame to redraw tile to display
void update()
{
    //m_Speed = 5 pixels per second
    //m_TimeInt = milliseconds elapsed since last frame
    //m_DirVec = direction vector

    float scalar = (m_TimeInt/1000) * m_Speed;
    CVector TmpVec = m_DirVec;
    TmpVec.scale(scalar);
    m_Position+=TmpVec;
}
```

7.7 Hierarchical Transformations

Q. Right; so the player (or other moveable tiles) can now move about the map, free from the rigidity of the map grid by using vectors to express directions and offsets, movements collectively termed *transformations*. I press the left arrow or right arrow and the player revolves to face new directions; I press the up arrow or down arrow and the player moves forward or backward, tracing along its invisible diagonal expressed by the LookAt vector, or direction vector. But there is a new problem: namely, collective movement, or *dependent transformation*. The tiles considered hitherto by this chapter have been single, self-contained units on the map such as the player, an NPC, or a car, each with an independent position and able to move about the map independently of one another. So long as this independence remains the case, transformation will work for each tile with no problems. But consider this example: There is a single map that contains three entities (*tiles*), each of them independent from one another insofar as each tile has its own position, LookAt vector, and speed in the map. There is a car tile, and seated inside the car is the player tile and an NPC tile. Now since each of the tiles — car and passengers — are positionally independent of one another, it means that as the car moves none of the passengers will follow since each tile has its own position. The car

simply will drive forward, leaving its passengers behind, literally. How can I solve this?

A. Until now each tile on the map has been conceptualized as an autonomous entity whose position, direction, and movement is largely independent of any other tile; this means that as one tile moves, another will not invariably follow because no relationship between the tiles is assumed to exist. However, the demands of most game maps are not so trivial as to allow relationships between tiles to be ignored. A relationship between any two tiles on a map is said to exist when the position (or orientation) of one tile depends upon (is *affected by*) the position of another tile, though this relationship need not be reciprocal. For example, the position of a tile X may reflect the changes in position of a tile Y, but tile Y may not be dependent on tile X. Thus, in any game map populated by many tiles there will be found a whole network (or *hierarchy*) of relationships between tiles such that no tile is completely independent of another. For example, all tiles standing on the ground are dependent on the ground; if the ground moves, so do the tiles standing on it, though the ground does not move with the tiles that walk upon it. In the same way, a car transports its passengers but the passengers do not transport the car; both the car and the passengers are dependent on the ground, however, since it is the ground upon which the car drives, and the car (*on the ground*) inside which the passengers are seated. This relationship can be expressed diagrammatically using the following hierarchy, a structure of relationship that can be found in all maps.

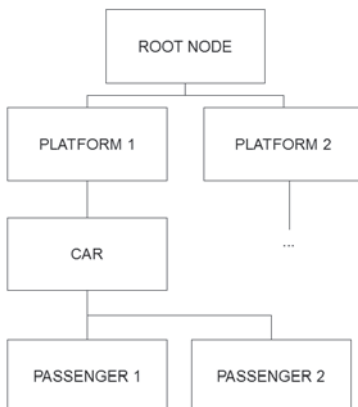


Figure 7-8: Relationship hierarchy

Hierarchical relationships refer to relationships of dependency, and as such the two popular terms “parent” and “child” are used to designate the influential partner and the dependent partner, respectively. The node whose position is affected by another is the child node, and the independent partner who affects the position of all its children is the parent node. In terms of the car analogy: The car is the parent of all its passengers (who are its children), but the car in turn is a child of the ground upon which it drives.

Support for hierarchical relationships can be implemented into the map tiles by amending `CGameObject` (the base class from which all tiles are derived) to include a `<vector>` list of child node pointers, listing all of its child nodes. This way, if every node is part (a leaf) of an overall hierarchical tree of parent-child relationships, all of whom trace their ultimate ancestor to a root parent at the origin of the map, then each node can define its X,Y position as an offset relative to its parent’s position rather than as an absolute offset from the origin of the map. The result is that as any node’s position in the map changes, it inevitably affects all its children and any generation of children beneath it in the tree throughout the hierarchy additively. The following code demonstrates this hierarchical additive transformation.

```
//Base game object
class CGameObject
{
private:
protected:
public:
    CVector m_Position;
    long m_Width;
    long m_Height;
    std::string m_Name;
    bool m_Visible;
    std::vector<CGameObject*> m_ChildObjects;
    CGameObject* m_Parent;

    CGameObject(std::string Name)

    {
        m_Name = Name;
        m_Position.x = m_Position.y = m_Width = m_Height = 0;
        m_Visible = true;
    }
}
```

```
        m_Parent = NULL;
    }

    virtual ~CGameObject()
    {
        m_ChildObjects.clear();
    }

    //[...] other functions here

    //Adds a child object to this tile
    void AddChild(CGameObject* Child)
    {
        m_ChildObjects.push_back(Child);
    }

    void update()
    {
        for(unsigned int i=0; i<m_ChildObjects.size(); i++)
        {
            //Recursively update through hierarchy
            m_ChildObjects[i]->update();
        }
    }
};
```

7.8 Z-Order and Depth Sorting

Q. The hierarchical tile arrangement easily solves almost all tile-relationship problems I can think of. Tiles can be children, parents, children of children, and so on throughout a whole hierarchy of tiles in a map. But still I have another awkward question for you, not related to the relationship between tiles as they are ordered positionally in the map, but related to the ordering of tiles as they appear on-screen, in the window. Specifically, the game window is the portal through which a gamer sees the contents of the game on every frame, in the same way a movie buff sees a movie from the perspective of the camera as projected onto the screen. This means that in any given map, some game tiles will inevitably be closer to the camera than other tiles, a

distinction between foreground and background. Consequently, tiles nearer the camera appear larger than more distant tiles, and nearer tiles also obscure any tiles directly behind them; for example, trees may partially obscure the sun setting behind them, or a player character may obscure those NPCs standing behind him. Let's call this problem "depth sorting." How can this best be handled?

A. This problem highlights the distinction between the position of tiles as they are generally arranged in the map in terms of X,Y position, and the *order* in which tiles are drawn to the screen. Indeed, the name itself, depth sorting, describes a process, and the name offers a clue to its solution. Depth refers to the depth order that any tile in a map occupies in relation to all other tiles and to the camera (e.g., this tree stands *in front of* this hill and *behind* the player). Sorting refers to the process of arranging or ordering tiles according to their nearness to the camera. In short, the depth sorting problem can be solved using the *painter's algorithm*, which works by first assigning to each tile in the map an integer (called its z-order) that reflects a tile's nearness to the camera. Tiles with lower z-orders are nearer to the camera than tiles with higher z-orders, which are farther away. Then on each frame, the algorithm proceeds to draw (render to the window) each tile according to its z-order, from highest to lowest. So distant tiles are drawn first, then closer tiles next, and so on, with tiles nearest to the camera being drawn last (on top of more distant tiles). Any tile, therefore, can be brought to the front of other tiles by lowering its z-order, or sent to the back of others (behind them) by raising its z-order.

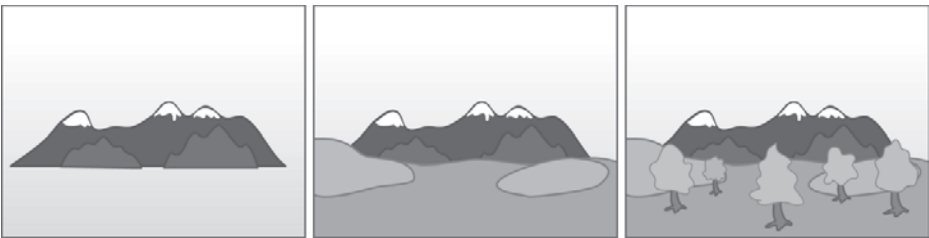


Figure 7-9: The painter's algorithm

The following code sorts and renders each tile according to its z-order:

```
void update()
{
    //Bubble sort order
    //Add property int m_ZOrder to CGameObject for index
    SortByZOrder(m_ChildObjects);

    for(unsigned int i=0; i<m_ChildObjects.size(); i++)
    {
        //Recursively update through hierarchy
        m_ChildObjects[i]->update();
    }
}
```

7.9 Conclusion

This chapter considered some of the most fundamental game mechanics that apply not only to almost all games across all platforms from Linux to Windows, but generally across both 2D and 3D games. Other game algorithms such as pathfinding, binary space partitioning, and octrees start from the premise that the game world is already represented as a Euclidean space where objects (tiles) are interconnected by spatial relationships to one another, and that each tile has a specified z-order that reflects whether it is nearer to or farther from the camera than other tiles. These algorithms may raise another, perhaps more reasonable question: In this technological age of instant communication, easy-to-use operating systems, and DIY/homebrew software, surely somebody, somewhere, has designed an easy-to-use game making package to create cross-platform games. Is there a package that doesn't require developers to reinvent the wheel by coding from scratch new pathfinding systems, depth-sorting algorithms, and other similar processes? Surely, the world of open-source software has a simple, easy-to-use, and powerful solution for making cross-platform games that "just work" without a lot of hassle. This topic is the focus of the next chapter.

This page intentionally left blank.

Chapter 8

Novashell and 2D Games

Games classified as 2D (two-dimensional) were written off by many gamers and critics as “dead and buried” the moment 3D games were born, but given the thriving 2D scene in contemporary gaming, those predictions join a long list of other industry predictions that thankfully failed to materialize. 2D games are alive and well, and testament to their health are famous titles from all gaming eras, such as Super Mario Brothers, Sonic the Hedgehog, Tetris, Dr. Lunatic Supreme with Cheese, Steam Brigade, and thousands more. Games classified as 2D are those where the action occurs in two dimensions only; that is, the gamer cannot move objects or cameras through a 3D gaming world to view objects from other perspectives. Thus, gaming perspectives are typically fixed in 2D games, and this makes them especially ideal for game genres such as platformers, puzzle games, side-scrolling beat-em-ups and adventures, and top-down blast-em-ups.

This chapter focuses on the development of cross-platform 2D games (Window, Linux, and Mac) using a free, open-source, and easy-to-use game development system called Novashell, which was created by independent game developer Seth Robinson. In a nutshell: Novashell is a click-and-drag style game development application for making cross-platform 2D games. The Novashell web site (www.rtsoft.com/novashell) describes the system as “a high-level 2D game maker that tries to handle all the hard work behind the scenes, allowing you to whip up sweet games using pathfinding, dialog, persistent dynamically sized maps with construction/deconstruction, save anywhere, and especially features that adventure and RPG type games would use.”

8.1 Novashell Overview

Freely available and based on the gaming API ClanLib, Novashell is an entirely integrated, self-contained, and GUI-based game development environment for creating cross-platform 2D games. Unlike most gaming software kits that depend on C++ IDEs such as Code::Blocks, Novashell instead works independently, and is all a user needs to start making professional standard games that run seamlessly across Windows, Linux, and Mac. Specifically, Novashell offers game developers the following key benefits:

- **Cut and Paste GUI** — Primarily mouse-driven, Novashell works much like an open-ended and extensible map editor, allowing developers to import art, animations, music, sound, scripted data, etc., and, by copying and pasting, assemble their data into coherent levels as a complete working game.
 - **Open-Source and Free to Create and Distribute Games** — Novashell as an application is freely available and is open-source, meaning it can be edited and extended to suit a developer's needs. Furthermore, developers are free to distribute their Novashell-made games as both freely available or commercial products.
 - **Scriptable** — Novashell allows developers to program and control their games through its integrated scripting system using an industry standard scripting language, Lua 5.1. Lua is a general-purpose, lightweight, C++-like scripting language used to edit and program games without recompilation. First created in 1993 by Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes from the Pontifical University of Rio de Janeiro in Brazil, Lua has been used in many commercial games including World of Warcraft, Grim Fandango, Supreme Commander, and Far Cry.
-

- **Online Documentation and Community Support** — Though Novashell is a relatively new GDK (game development kit), first released in early 2007, it already has gained a sizeable online forum community of game makers, and also features comprehensive online documentation featuring references for scripting (classes, functions, and objects) and also “quick-start” guides for getting up and running with Novashell.



NOTE. The following is a brief list of some of the additional features of Novashell:

- Multi-platform support for Windows, Mac OS X, and Linux
- Open-source under a zlib/libpng license
- Hierarchical goal-based AI system
- A* (A-star)-based pathfinding for navigating an unlimited number of connected maps, warps, and doors
- Free-form pixel-accurate sprite patch editing as well as emulated conventional tile editing
- Map dimensions can flexibly change during play, and new areas can be added anywhere at any time
- Robust automatic save/load allows a fully persistent world
- Built-in world editor with unlimited brush sizes, multiple undo, and cut and paste of any size
- Parallax scrolling
- Powerful particle system
- Real-time “smart shadow”
- Able to create stand-alone games

8.2 Downloading Novashell (Windows, Linux, and Mac)

In summary, Novashell is a free, open-source, and experimental tool allowing developers to create cross-platform 2D games easily via a WYSIWYG interface. Both the Novashell source code and the Novashell binary executables for Windows, Linux (Ubuntu), and Mac are freely available from the official Novashell web site at <http://www.rtssoft.com/novashell/>. The procedure for downloading and installing Novashell, which is the same for all supported platforms, is described step by step as follows.



NOTE. Like many open-source projects, and especially those in their infancy, the Novashell web site defines Novashell as “not feature complete,” or as an experimental application. It is to be considered as an open-source work in progress continuously being refined and enhanced as time goes by.

1. Beginning from the (Windows, Mac, or Ubuntu) desktop, open a web browser and navigate to the following address:
<http://www.rtssoft.com/novashell/>.

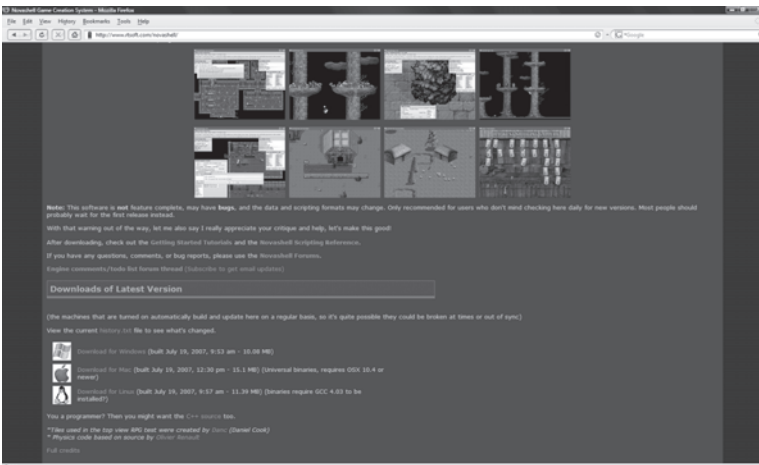


Figure 8-1

2. Click the download link appropriate for the host operating system and save the archive to any directory on the local computer. For Windows only, users should run the packaged Novashell installer to install Novashell to the system.

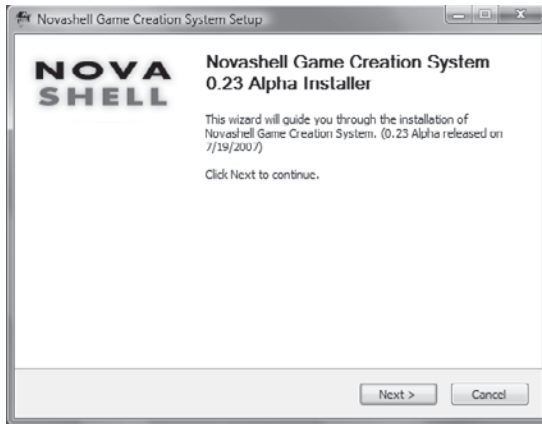


Figure 8-2

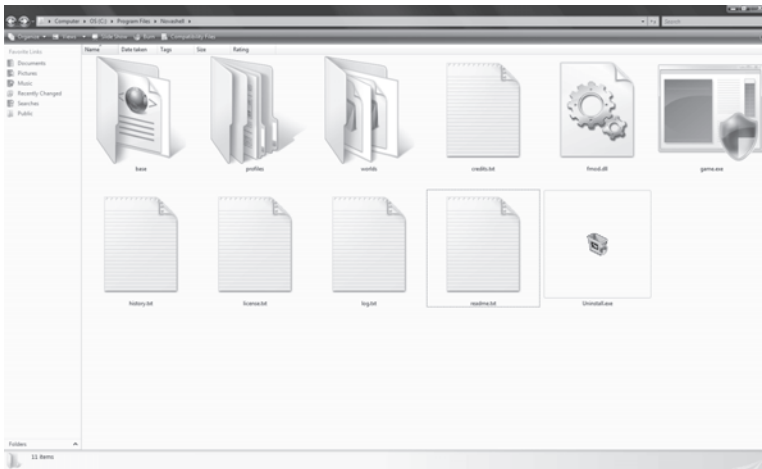


Figure 8-3

The downloaded and extracted package (or the installed files on Windows) contains the following important files and folders:

- **Game.exe** — The main Novashell executable; features the integrated game editor for designing and creating games, and also doubles as the interpreter (or virtual machine) for playing Novashell games.

- **History.txt** — Records the developmental progress of Novashell from its first release until the current downloaded release, detailing bug fixes for earlier editions and newly added features present only in the current release.
- **Base directory** — Features all global data (graphics, sound, scripts) used by Novashell for its editor and for all Novashell-developed games. Data featured in the base directory applies globally to all games, and not specifically to any one game.
- **Worlds Directory** — Game-specific world data (graphics, sound, scripts) for a single Novashell game. Each game houses its game-specific data within any appropriate nested subdirectories of the Worlds Directory; more on this later.

8.3 Exploring Novashell Games

Novashell ships with a handful of semi-completed sample games to play, each intended to demonstrate the variety of Novashell features and the diversity of 2D genres that it supports — from 2D side-scrolling games to top-down RPGs and space-invader-style shooters. These games include Tree World, Beer Invaders, Dink RPG, and Tanks, and are considered briefly below.

- **Tree World**



Figure 8-4: Tree World

One of the highlighted sample games is a side-scrolling platformer called *Tree World*, which demonstrates the simplicity of parallax scrolling by using Novashell and also shows off its particle systems and collision-based physics. Much of the scripting for this game can be found in a Lua text file located in the *Worlds* subfolder of the Novashell root directory in a Lua script called `ent_player.lua`. The objective of the game *Tree World* is to explore the levels, fight or avoid enemies, and collect as many coins as possible.

■ Beer Invaders



Figure 8-5: *Beer Invaders*

Based on the famous arcade classic *Space Invaders*, the Novashell derivative, *Beer Invaders*, substitutes beer cans for spaceships. Each beer-can-enemy appears at the top of the screen and can in turn fire smaller beer cans downward as bullets. The objective is to slide a pistol firearm horizontally along the bottom of the screen and shoot all approaching enemy cans without getting shot by the bullets. This tricky game primarily demonstrates collision detection between bullet and target, but also demonstrates music, sound, sprite animation, and goal-based levels.

■ Dink RPG



Figure 8-6: Dink

The sample Dink-style RPG (role-playing game) that is based on the Dink Smallwood game by Seth Robinson demonstrates clearly in one contained sample — perhaps more so than any of the other included sample games — the wide subset of features Novashell offers to a developer for creating 2D games. This RPG allows characters to warp in and out of different maps across the game world, to converse and reply intelligently to each other, and to engage in combat with enemies scattered about the world.

■ Tanks

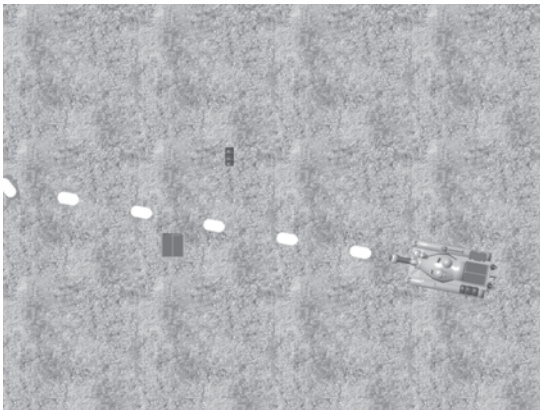


Figure 8-7: Tanks

Tanks is a small but important sample game in which the player controls a military tank. The player may shoot or push nearby

objects, which in turn react and move according to the scripted physics.

8.4 Getting to Know Novashell

Novashell is designed both to create 2D games and to play (or execute) the 2D games it creates, in the same way a Java virtual machine runs Java applications or in the same way the Flash Player plays Flash presentations. Consequently, Novashell is not entirely unlike Flash, but whereas Flash considers the player and the editor to be separate entities, Novashell condenses them into an integrated player and editor. Thus, Novashell stands apart first and foremost as an application that is both a game player (or *executor*) and a game editor — a game player in that Novashell games can be opened and played by the Novashell application and a game editor since loaded games can be paused and edited. However, compiled stand-alone games (games that run independently of the editor, primarily so that players cannot cheat easily) is one feature among many of the planned features along the Novashell developmental road map still to be implemented. Novashell as both a player and an editor is now examined more closely, beginning from the startup screen that appears immediately after running Novashell (on any platform).

8.4.1 The Game Selection Menu

Novashell begins at the Game Selection menu, a screen from which both gamers and developers choose from among the available Novashell games installed locally to play and/or edit. By default this menu features at least the sample games listed in the previous section, and gamers may double-click any game from the list in order to play or edit them. (Pressing the Esc key at any time exits the game currently being played and returns to this menu.) Broadly, each game listed on the Game Selection menu corresponds to an individual config file located in the Worlds subdirectory of the root Novashell installation directory. Each file there lists the details of a specific Novashell game,

such as game name, valid directory paths to graphics, sounds, scripts, etc. We'll discuss these files in a later section.

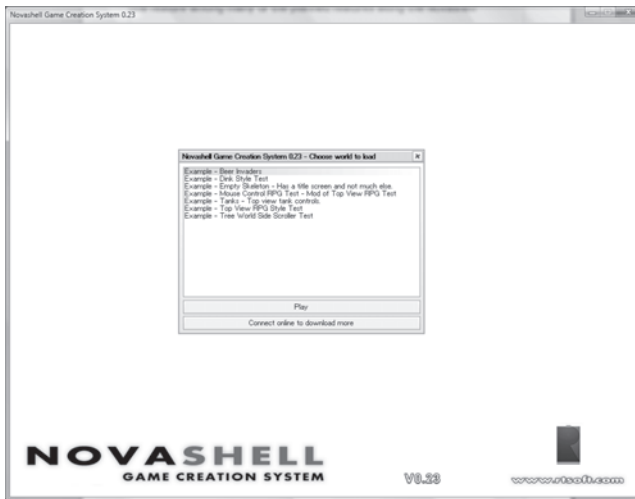


Figure 8-8



NOTE. Ctrl+Alt+Enter switches Novashell between full-screen and windowed mode.

8.4.2 The Editor and Player Modes

Novashell may work in one of two modes at any one time: editor or player mode. These two modes appeal to developers and gamers, respectively. Pressing the F1 key toggles between the modes. Novashell in editor mode offers the required tools and facilities to change and edit existing games while those games are paused; developers can even scrap games wholesale, rebuilding them again from scratch. Novashell in player mode hides its editing facilities, resumes a game that is paused, and allows both gamers and developers to play and/or debug their Novashell games in real time just as though the game were compiled to run stand-alone.



NOTE. Novashell can be switched between player and editor mode at any time by pressing F1.

8.4.3 Getting Started – Loading, Playing, and Editing a Game

Novashell offers facilities to load, play, and edit games, and this section examines how to load and play a game. The Novashell world editor is discussed in the following section.

1. Start Novashell and double-click the game **Dink Style Test** from the Game Selection menu.
2. The game menu then offers three options: New, Continue, and Edit. Click **New**. These options apply across all the default Novashell games and are important, as considered below.



Figure 8-9

- **New** — This option begins play of the selected game from the beginning, as defined by the developers. Pressing F1 to start the editor during any games run from the New option allows developers to make changes only to the *currently running* instance of the game (changes which are forgotten upon exiting the game), as opposed to permanent changes made globally to all future instances of the game played from New (for this, select Edit instead).
- **Continue** — This option resumes play of the selected game from the point where any previous play had ended.

- Edit** — The Edit option opens the game for editing globally. That is, any changes applied to a game in Edit mode will apply permanently, and not temporarily, to all future instances of the game. Editing in this mode is like editing an original photograph instead of a copy, or editing the template of a letter instead of the letter itself.
3. Having clicked New to play the game, press **F1** to pause the game and to display the Novashell game editor, which is described in the next section.

8.5 Novashell Editor

The Novashell editor offers an abundant set of tools (palettes, layering, collision detection, scripting, copy and paste) for making practically



Figure 8-10

any conceivable changes to Novashell games. These tools range from graphic import facilities to map designing facilities such as tile editing. The raw materials external to the game itself and from which game data is based such as image files, sound files, and script files are collectively termed *resources*. It is from the synthesis of those resources, the bringing them together into a coherent totality, that games are made (Novashell games included). Conceptually, resources are imported into

Novashell games as either a *tile* or an *entity*, depending on the purpose they are to serve for the game.

8.5.1 Tile Resources

The graphical term “texture” refers to any graphical resource (any image, from PNG to BMP, including transparency information). Textures are typically loaded from a file and into Novashell as tiles (the simplest graphical entity featured in a Novashell game). Tiles are textures and are usually designed to be small and lightweight since they are likely to be copied and pasted many times in various combinations and arrangements to build a single Novashell level (for trees, floor tiles, walls, doors, windows, etc.). Often, tiles are not imported separately into Novashell from individual textures (with one file per tile). Instead, developers import a single, larger texture onto which all tiles for a single level have been collectively arranged in rows and columns (called a *texture tile set*), and these tiles are then cut out from the tile set subsequently as separate tiles using the Novashell editor. Tiles have the distinct advantage over entities in being simple to use, versatile, and “lightweight” (meaning they are computationally efficient with low memory costs). Specifically, tiles boast the following features:

- **Flipping and Alpha** — Each tile in Novashell may have alpha transparency; that is, some pixels of the tile can be fully transparent, semi-transparent, or wholly opaque. A tile’s transparency data is defined by its texture information (for those file formats that support transparency). Each tile may also be flipped by the Novashell editor; that is, a tile’s pixel data can be reflected (mirrored) about its central x or y axis, reversing the image symmetrically. This is a useful technique for giving some variability to tiles that are frequently repeated throughout a single level, such as floor tiles.
- **Multiple Instances, Global Types** — Any single tile may be copied from the original and pasted in a Novashell level any number of times, each new copy itself becoming an individual tile. For example, an initial tree tile may be duplicated many times across a level, creating a level with many trees, and each pasted tree then becomes a separate tile belonging to the type “tree.” Specific changes to the tile’s type (changes in collision detection, changes

in graphical appearance, etc.) apply globally across all tiles in a level.

8.5.2 Entity Resources

Entities are extended, or advanced, tiles. In a nutshell, entities exhibit all the characteristics of tiles, but boast additional features and are typically used for more complex level objects, such as moveable game characters and NPCs (non-player characters), interactive game scenery such as moveable platforms, or collectable power-ups. The basic Novashell rule of thumb may be: “If it’s a moveable or interactive object, then it’s an entity; if it doesn’t move and isn’t interactive, then it’s not an entity; and if it’s not an entity, then it’s a tile.” In Novashell, like in most games, most things are tiles and not entities. In addition to the properties of tiles, entities also boast the following features:

- **Scriptable** — Any entity may have a Lua script attached to control its behavior and receive event notifications, which are function calls a Lua script receives whenever important events occur to an entity, such as when an entity collides with another entity or tile in a level, or when the player presses a button on the keyboard.
 - **Unique** — Unlike tiles where any single tile type (such as a “tree”) shares its collision data between all tile instances of that type in a level, entities of the same type may each have unique collision data.
 - **Visual Profiles** — Entities may be assigned a whole series of different frame-based animations for particular “states.” For example, entities like player characters and NPCs may require a specific animation when in a walking state and another animation when in a fighting state. Novashell offers externally written XML files (examined later) called visual profiles that can be assigned to entities, and each profile details the frame-based animation to be played during different entity states.
-

8.6 Novashell Tools

Novashell levels are a collection of tiles and entities, and the Novashell editor is where developers create that collection in which tiles and entities (player, walls, doors, floors, etc.) are arranged to form a game world. At this point, readers should have started a new Dink Style Test game from the Novashell Game Selection menu, and then pressed F1 to pause the game and switch to the Novashell game editor mode. Let's take a look at the editor's tools and windows.



TIP. To get the most from this chapter, it is recommended that readers follow along page by page while also using Novashell.

- Main Menu** — The Novashell main menu provides access to many of the core features, from exiting Novashell to pausing/unpausing games, and also for saving the edits made to each game. These menus also provide options to show and hide hidden game data, such as collision and pathfinding information.

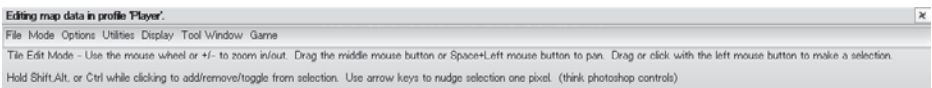


Figure 8-11: The main menu

- Map Switcher** — Every Novashell game contains a collection of maps, and each map is in turn a collection of tiles arranged to make a level. The Map Switcher window lists every map currently in the active game.

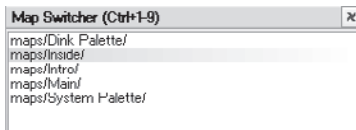


Figure 8-12: The Map Switcher window

- Tile Editor** — The Tile Edit window allows developers to perform a variety of functions. From this window, you can import game art as tiles and entities; edit individual tiles and entities, such as collision information and script

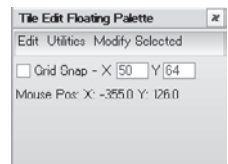


Figure 8-13: The Tile Edit Floating Palette window

data; and edit Novashell maps, changing their grid layout and other options.

- **Layer Editor** — The layers in Novashell work similarly to those in photo editing applications like Photoshop and GIMP. Like the clear plastic layers stacked atop one another and used by animators to make cartoons, the purpose of layers is to define the order in which tiles and entities in a level are drawn to the window in order to create a sense of depth. The process of arranging the depth order of tiles is known as *z-ordering*, where each tile or entity in a level must be assigned to only one layer, and each layer determines the order in which tiles are drawn to the window. Tiles assigned to the foreground layer (the topmost layer) will be drawn *after* tiles assigned to the background layer. Thus, foreground tiles appear in front of background tiles, and background tiles are obscured by tiles in front of them that occupy the same X,Y screen space. The three exceptions to this rule are the entity layer and the two hidden layers. The *entity layer* is used for housing entities rather than tiles because entities, unlike tiles, can move and change their position dynamically (such as NPCs), so their z-order in relation to one another can change at any time. Consequently, the entity layer is designed to be “smart,” and it automatically calculates which entities are in front of or behind others based on their X,Y positions and draws them in the appropriate order on every frame. The *hidden layers* are designed for housing non-visible level data, such as music and other resources.

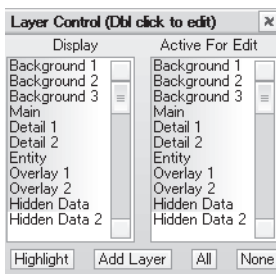


Figure 8-14: The Layer Control window

8.7 Editing Novashell Levels

This section explores some common features of the Novashell editor for creating and editing game maps.

8.7.1 Selecting, Copying, Pasting, Moving, and Filling Tiles

Editing maps in Novashell is GUI-based, and so levels are created by copying and pasting tiles and entities by using the standard Ctrl+C (copy) and Ctrl+V (paste) keyboard combinations, or by selecting any tile or series of tiles and right-clicking the mouse to display a contextual menu offering copy and paste options. Some other options are described below.

- **Copy and Paste** — Selecting a tile, pressing Ctrl+C to copy the tile to the clipboard, and then pressing Ctrl+V to paste new tiles causes the new tile to appear at the X,Y location of the mouse cursor. Tiles and entities may also be copied and pasted *between* maps.
- **Flood Fill** — To fill a region with a specific tile, first copy a tile to the clipboard, then press and hold the Shift key while using the mouse to draw a selection box over a region of the map. Press F to Flood Fill the selected region with as many new instances of the copied tile as the region may hold.
- **Cut, Undo, and Deselect** — Ctrl+X cuts rather than copies tiles, Ctrl+D deselects all (selects nothing) in the map, and the standard Ctrl+Z (undo) combination reverses the last edit (if any) made to the map.
- **Scaling** — The left and right bracket keys ([and]) scale the selected tile(s). [decreases the scale for as long as the key is pressed, making the selected tiles smaller, and] increases the scale, making them larger.

- **Select All of Kind, Additive Select, Subtractive Select** — Selecting a tile and pressing Ctrl+R selects all other tiles of the same kind in the current map, that is, all other tiles copied and pasted from the same original. Press and hold Ctrl while selecting to add further tiles to an existing selection, and press and hold Alt to deselect tiles in an existing selection.

8.7.2 Exploring Maps and Editing Tiles

This section explores both how levels are navigated using the Layer Editor window, and the basics of the Tile Properties window.

- **Level Navigation** — Pressing Ctrl+(0-9) takes you to the first 10 maps; you can also use the standard map selection window to switch between maps. The + and – keys (or the mouse wheel) zoom in and out of levels so you can examine them up close or from a distance. Space+(click and drag) scrolls across the level.
- **Tile Properties** — Open the Tile Properties window by right-clicking a tile and choosing Properties from the context menu. The Tile Properties window displays a series of editable, common properties for the currently selected tile in the map.

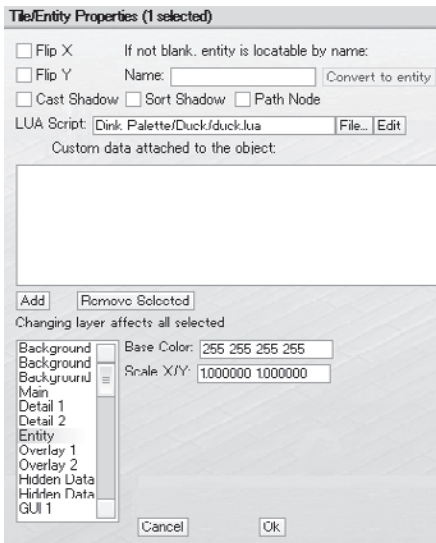


Figure 8-15: The Tile Properties window

The Tile Properties window displays most of the editable properties for the currently selected tile or entity in a Novashell map. The check boxes for Flip X and Flip Y mirror (or reflect) any tile or entity about its axis of symmetry. The Cast Shadow check box causes Novashell to calculate and generate the shadow shape that the selected tile casts on the surrounding map, based on its collision and pixel data. You can also assign the selected tile to a specified layer to control its z-order. Click the Convert to entity button to convert the selected tile into an entity (non-reversible). The LUA Script field allows you to attach a valid LUA script defining the entity's behavior.

8.8 Creating New Games and Maps

A Novashell game is a collection of levels, or perhaps more correctly, a collection of maps. All prior sections in this chapter considered only how to edit existing games and existing maps using the Novashell editor. In the following steps, we'll explore the process of creating a new Novashell game from scratch starting from an empty project.

1. Beginning from the desktop, open the local folder where Novashell was extracted or installed. This folder features the Worlds subfolder, which contains additional subfolders that each correspond to a unique Novashell game and contain the game resources (the graphics, sound, data, script files, etc.) specific to that game. The Worlds folder also contains .novashell config files for each individual game. Each .novashell config file details in a simple line-by-line text format the core properties of a unique Novashell game, such as game name, author name, resolution, etc.
2. Newly created Novashell games are usually created from the empty skeleton project template called RT_EmptySkeleton, located in the Novashell Worlds folder. Its.novashell file is RT_EmptySkeleton.novashell. To create a new project, make a duplicate of this folder and this file and rename each of them to reflect the name of your new game.

3. Open the newly created `.novashell` config file using a standard text editor application and edit each field in the file as appropriate for the new game. Enter a name for the game title, the desired resolution, etc., as illustrated in the following sample `.novashell` file:

```
//this file contains important data about the world that the
//engine checks before it loads anything

//world will fail if the engine is older than this. Newer
//versions will attempt to emulate this version
engine_version_requested|0.22
world_version|0.1
world_name|My Test Game
world_description|This is my first game
world_author|Me
world_website|www.alanthorn.net
desired_resolution|1024|768|32

//must be 200X200 jpg
world_thumbnail|

//if this is a mod of a mod(s), add its dependencies here (dir
//name|version required) (base is assumed, don't need to add
//that)
//add_world_requirement|SomeModA|0.0
//add_world_requirement|SomeModB|0.0
```

4. Having duplicated both the `RT_EmptySkeleton` project folder and its corresponding `.novashell` config file, and having tweaked them in preparation to become a new Novashell game, start the Novashell editor. The newly created game appears on the Game Selection menu. Select the game and switch to Edit mode to play and edit the master files.
-



NOTE. After being selected from the Game Selection menu, a Novashell game begins by searching every one of its maps for an entity named “player,” which is an entity to be controlled by the gamer and where the game camera first focuses as the game begins. The player entity represents the topological point at which a game begins. Games created from the Novashell skeleton project do not, however, feature a player entity by default since not all games require a player entity. In this case, an error message may appear at startup to notify the developer that no entity named “player” exists in any map for this game. This error can be ignored by clicking OK, and a player entity can be added at some later stage, as demonstrated shortly.

The game is now ready to edit.

The following sections of this chapter explain how to import art, select a player entity, implement Lua scripting, add collision detection, and create some basic AI (artificial intelligence).

8.9 Importing Art into Novashell

Newly created Novashell games based on the skeleton project begin their life as a completely empty shell — every map is without graphics and without sound. This section explores how the Novashell editor is used for importing graphical tile sets from files on disk and into Novashell as either tiles or entities.



NOTE. Before starting the game, all required image files should be copied into the current game’s folder, which is a subfolder of the Novashell Worlds folder.

8.9.1 Importing Files

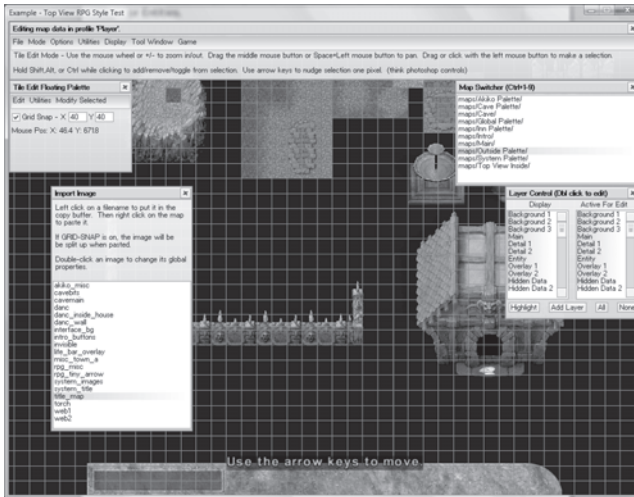


Figure 8-16: Importing image files

Importing image files into Novashell means streaming all pixel data from the files and loading them in Novashell into a memory buffer. This process is achieved using the Tile Edit window as follows:

1. Click the **Grid Snap** check box in the Tile Edit window, and set the grid size to between 40 x 40 and 100 x 100. Grid snap locks tile movement to predefined grid spacings in the map, and is a useful tool for level editing.
2. From the main menu, choose **Utilities | Import Image(s)**.
3. The Import Image window appears, displaying a list of valid graphic files available for importing. Click to select a file from the list, and its pixels are transferred from disk and onto the Novashell copy buffer, ready for pasting into the current map as one self-contained tile.
4. Press **Ctrl+V** to paste the copied pixels onto the map as a tile. Repeat steps 1-3 for each image to be pasted.

8.9.2 Setting a Player Entity

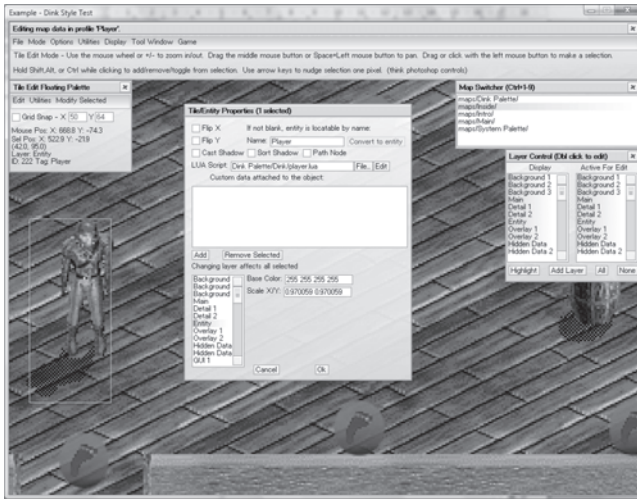


Figure 8-17: Creating a player entity

Entities are sophisticated tiles. For games where the gamer controls a character, it is standard practice in Novashell to create a *player entity*, the tile representing the gamer-controlled character. To create an entity using the Novashell editor, right-click a tile to display the context menu for the selected tile, and click the Properties option to show the Tile Properties window. Click the Convert to entity button to convert the tile to an entity. To make this entity the player entity, enter Player for the entity name.

8.9.3 Creating Smaller Tiles from Larger Tiles

Each graphic resource is imported from a file on disk and into Novashell first as a complete, self-contained tile (not entity) using the Tile Edit window. This is true even if the file is arranged in rows or columns as a tile set and is intended to represent a palette of separate tiles, such as a file containing 40 x 40 pixel rectangles for floor tiles, wall tiles, NPC tiles, etc. For tiles of this kind, the Novashell editor features a series of cutting tools to visually “cut,” or subdivide, a larger complete tile into a set of constituent, smaller tiles independent of the larger tile. The following steps explain the Novashell tile-cutting process more completely.

1. Beginning in the Novashell editor with a single and complete imported tile, hold down the **Ctrl** key and drag a selection over the rectangle to “cut” from inside the larger tile.
2. Before releasing the mouse button, press **Ctrl+C** to copy the selected rectangle of pixels to the copy buffer.
3. The selected pixels copied to the buffer can now be pasted as independent tiles using the standard paste shortcut, **Ctrl+V**.

8.9.4 Setting Collision Information

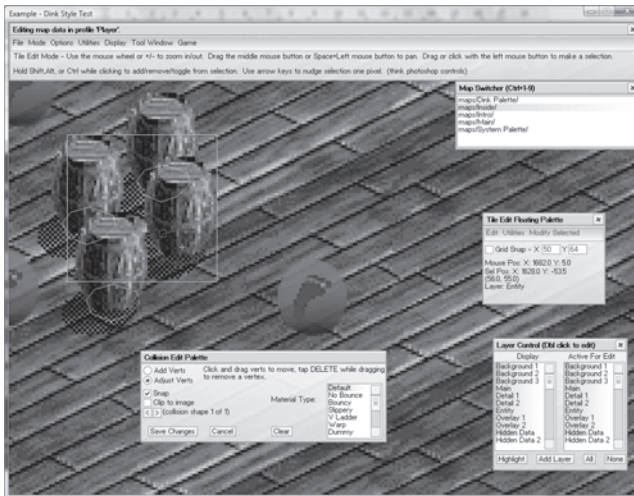


Figure 8-18: Setting collision detection

Traditionally, computer games are designed to simulate real-world environments insofar as game characters cannot walk through walls or effortlessly pass through obstacles and enemies without sustaining damage. Thus, any two game objects (solid bodies) whose boundaries intersect one another at any one time are said to *collide*, meaning they contact one another. The process of determining when collisions between objects occur in a game is called *collision detection*. Novashell detects collisions automatically, and it is the role of the developer only to indicate the borders of tiles so Novashell may determine when the border of one tile intersects the border of another. The following steps illustrate how to create a collision border for any selected tile in the Novashell editor.

1. Beginning from the Novashell game editor, select a tile or entity for attaching collision information.
2. From the Novashell main menu, select **File | Modify Selected | Edit Collision Data**. The Collision Edit Palette appears.
3. From the Collision Edit Palette, choose **Add Verts**.
4. Draw a shape (collision boundary) around the tile by clicking the mouse to add new vertices.
5. Click **Save Changes** in the Collision Edit Palette when completed. Repeat these steps to set the collision information for each tile, as required.



NOTE. To receive run-time notifications from Novashell whenever collisions between tiles occur in the map, scripts should be added to each entity involved in collisions. Scripts are examined in more detail later in this chapter.

8.10 Novashell System Palette



Figure 8-19: The System Palette

Novashell levels created in this chapter thus far have featured visible tiles only; that is, tiles that actually appear in the game window. In

addition to importing visible tiles and creating entities from those tiles, like the player entity, Novashell levels may also feature special invisible tiles, designed for specifying and controlling level-specific behavior. These tiles belong on the hidden layer where they exist spatially alongside all other tiles in the level, both visible and invisible. Such special tiles may specify, for example, which music is to be played while the player moves inside the tile's boundary, or they may initiate a Lua script that is to be run at specified times or when a player enters a room or crosses the threshold of a doorway. The Novashell editor features a special palette of such tiles listed together in the System Palette, where developers may go to copy its available tiles and paste them on their standard maps just like regular tiles, editing the properties of each tile instance as suitable. The System Palette features many special tiles, including audio, color, invisible wall, warp, waypoint, path, and script.

8.10.1 Audio Tiles

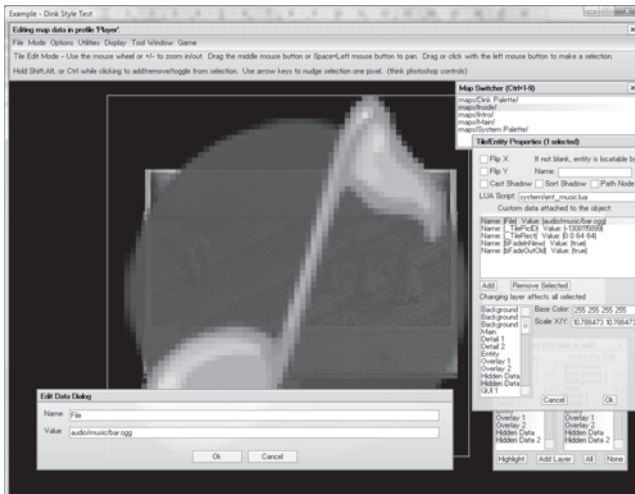


Figure 8-20: Placing an audio tile

The Novashell System Palette features five audio tile types available to copy and paste on the hidden layer of any standard Novashell map. The audio tiles play (or stop playing) a specified sound or song whenever the player character enters the radius of its circular area; that is, whenever the X,Y position of the gamer in a level is nearer to the

circle's center than the extent of its radius. Audio tiles remain dormant in a level until a player enters their circular region. The Ambience Loop (crossfade) tiles merge or blend any previously playing song into a specified song. The Music (instant) tiles immediately cut the play of any prior songs and begin playing a specified song. The Music (fade out) tiles reduce the volume of any current playing song down to silence.



NOTE. Like regular tiles, audio tiles can be scaled to smaller and larger sizes using the [and] keys.

To create a music tile, set its radius, and play a song, perform the following steps:

1. Beginning from the Novashell System Palette, select an Ambience Loop (crossfade) audio tile and copy it to the clipboard.
2. Paste the audio tile into any of the game's maps, then open the Tile Properties window from the right-click context menu.
3. Double-click the Name field in the Custom data attached to the object list, and the Edit Data Dialog window appears. Enter a valid path to an audio file in the Value field. Acceptable formats include MP3, OGG, WAV, ASF, S3M, and others. Click **OK** when completed.
4. To change the audio tile's radius, select the tile and use either the [key or the] key to shrink or enlarge the tile as appropriate.

8.10.2 Color Tiles

Color tiles are the only tile available from the System Palette that also show in the game map rather than remaining hidden like audio or AI tiles. Color tiles appear on the game map as a bold, programmer-defined color, and are useful for creating a quick and dirty mock-up in the absence of game-ready artwork.

8.10.3 Invisible Wall Tiles

As the name suggests, Invisible Wall tiles are used first and foremost as blocks or obstacles to prevent gamers moving past them. They can also be used to simulate invisible barriers and force fields.

8.10.4 Warp, Waypoint, and Path Nodes

Warp, Waypoint, and Path nodes together constitute Novashell’s “pathfinding” framework — the system of invisible map markings and connected graph nodes allowing NPCs and other “intelligent” characters to find their way around maps, plan routes, and avoid obstacles. A Warp file marks the X,Y position of an entrance or exit point on a Novashell map so that a Warp on a different map can allow game characters to move between those maps, leaving via the exit point on one map and arriving to the specified map at the entrance point. Waypoint and Path nodes work together insofar as they help NPCs move sensibly from X to Y in any map, avoiding obstacles and other problems. Path and Waypoint nodes are invisible X,Y markers placed by developers throughout a level to build a connected network. Using this network of connected points, NPCs calculate and plan routes when required to travel any distance on a map.

8.10.5 Script Tiles

Novashell works with Lua scripts. Any Novashell map may feature scripts attached either to entities in a map or to system palette tiles. The next section considers scripting more closely.

8.11 Novashell Scripting

Though Novashell features no integrated text editor, it is primarily a script-driven engine, and all Novashell games are coded using Lua scripts. Lua is a lightweight and extensible scripting language whose syntax bears some resemblance to C++; this makes it easy to pick up and start using Lua to create Novashell games. Scripts are distinguished from compiled languages like C++ in which source code is first written and then compiled into executable form. Scripted languages are instead an *interpreted* language whose source code (script) is loaded from text files and then compiled, or executed, by Novashell on a line-by-line basis at run time. This difference in approach between compiled and scripted languages means scripts can be opened in a

standard text editor and amended to make changes in the game without a need to recompile. Scripts in Novashell games are attached to entities. These entities can be standard entities like the player or NPCs as well as System Palette entities on the hidden layer of a map. Scripts can also be executed interactively from the Novashell Console, which we'll discuss next.

8.11.1 Novashell Console

As featured in Quake 3 and Unreal, the Novashell Console is an interactive panel where system messages are shown and where Lua scripted commands can be executed at run time. From here, scripts can output debug messages, print error notifications, and return confirmations, and developers can run commands, programmatically edit the game, and execute specified scripted functions. Press the single quote (') key to toggle the Console on and off. Following are some simple tasks that can be accomplished using the Console.

- To select an entity on the map by name using the Console, type the following and then press Enter (this example uses the `GetEntityByName` function):

```
GetEntityByName("Player");
```

- Each entity with a script attached can be thought of as an object-oriented class. These entities feature a set of scripted methods and properties such as the `Run()` and `Walk()` methods and `Health` and `Speed` properties. To list the script data including its attached properties and methods for a selected entity, enter the following into the Novashell Console and then press Enter:

```
this:DumpScriptInfo();
```

- To run any of the scripted functions listed by the `DumpScriptInfo` command for any selected entity, type the command in the following form and press Enter:

```
this:RunFunction("FunctionName");
```

Where `RunFunction` is the specified method of the selected entity, and `FunctionName` is the name of the method to run.

8.11.2 Attaching a Script to an Entity

The Novashell Console is interactive insofar as it allows developers to initiate functions manually from the command line, and allows the initiated functions to print their debug messages back to the window in response. In addition to the command line, script files can be attached to specific entities to handle entity events and behaviors. Entity events include mouse clicks, keypresses, collision events, `OnMapBegin`, and `OnMapEnd`. Generally, scripts are written to handle and control an entity's response to these events, where one event typically corresponds to one method in the Lua script file (an event handler), and this handler is initiated at run time on each occasion the corresponding event occurs to that entity. The following example demonstrates the step-by-step process for creating a new Lua script file and then assigning that file to an entity named "Player."

1. Beginning from the desktop, create a new text file using a standard text editor or a Lua editor such as LuaEdit (<http://luaedit.luaforge.net/>).
2. Enter the following basic script, handling the three simplest events for any entity. These event handler functions are described after these steps.

```
RunScript("system/player_utils.lua");

function OnInit() //run as game is executed and entity created
    this:GetBrainManager():Add("StandardBase","");
    this:SetIsCreature(true);
end

function OnPostInit() //run once entity appears on map
    LogMsg("The entity name is " .. this:GetName());
end

function OnKill() //run as entity is removed from memory
    RemoveActivePlayerIfNeeded(this);
end
```

3. Save the Lua file with a unique name (e.g., “newplayer.lua”) in the Worlds subfolder appropriate for the game.
4. Load up Novashell and edit the game. Add a player entity to the map and right-click on the entity to display its context menu. Click **Properties** to open the Tile Properties window, and enter into the Lua Script edit box the fully qualified path to the newly created Lua script file. The script assignment is now completed.

This sample Lua code features three of the most fundamental event handlers a Novashell script can contain for an entity: `OnInit`, `OnPostInit`, and `OnKill`. The `OnInit` handler is executed once per game at the moment the entity is created in memory. `OnPostInit` occurs after `OnInit`; it is executed on every occasion the entity is placed on the map. `OnKill` is run once per game and occurs whenever the player is scheduled to be deleted from memory; this is typically as the game is terminated by the player. These three functions are now considered more closely.

```
function OnInit() //run as game is executed and entity created
    this:GetBrainManager():Add("StandardBase","");
    this:SetIsCreature(true);
end
```

- **GetBrainManager** — A standard Novashell function that should be called once in a script file for every intelligent entity on a map. An *intelligent entity* is one that can move in terms of its X,Y location on a map. This function allows an entity to exhibit basic intelligence, configuring the entity to react to collisions realistically and to find its way through a map across a series of plotted path nodes whenever it is instructed to do so.
- **SetIsCreature** — This function flags the entity as a moveable, intelligent entity in a Novashell map; the pathfinding system will not figure this entity as a static obstacle for other entities when performing its pathfinding calculations.

```
function OnPostInit() //run once entity appears on map
    LogMsg("The entity name is " .. this:GetName());
end
```

- **LogMsg** — This function prints a specified debug string to the Console window.

```
function OnKill() //run as entity is removed from memory
    RemoveActivePlayerIfNeeded(this);
end
```

- **RemoveActivePlayerIfNeeded** — This function flags to Novashell that the specified entity may be safely deleted from memory since it is no longer required for the game.

8.11.3 Visual Profiles

Any single entity on the map, such as the player entity, may be in one among many different states (such as walking, running, jumping) at any one time. The player entity may, for example, be in a standing-still state or in a walking or running state, depending on which keys the player presses. Consequently, the visual appearance of an entity will probably change depending on its state, requiring a different graphic or animation to represent the entity in each state. Novashell uses its Visual Profiles feature to solve this problem. A *visual profile* in Novashell is an XML file assigned to an entity that lists each entity state by name (“walk,” “run,” “jump,” etc.), and associates a visual style (an appearance) to each state. Once the developer has associated a complete XML profile to an entity and listed every state that entity may assume, he then programs (in Lua) the entity’s currently active state at run time from among the list of states available in the profile. This is a bit like a dial or switch on a washing machine for choosing which mode to work in. An example visual profile is featured below.

```
<resources>
<profile name="main_character">
  <anim state="idle_left" spritename="idle_left" mirrorx="no" />
  <anim state="idle_right" spritename="idle_left"
    mirrorx="yes" />
  <anim state="walk_right" spritename="walk_right"
    mirrorx="yes" />
  <anim state="walk_up" spritename="idle_down" mirrorx="no" />
</profile>
<sprite name="idle_left">
  <image fileseq="idle/idle_left.png" leading_zeroes="3" />
```

```

    <translation origin="center" />
    <animation pingpong="no" loop="yes" speed="150" />
</sprite>
<sprite name="idle_right">
  <image fileseq="idle/idle_right.png" leading_zeroes="3" />
  <translation origin="center" />
  <animation pingpong="no" loop="yes" speed="150" />
</sprite>
<sprite name="walk_right">
  <image fileseq="idle/walk_right.png" leading_zeroes="3" />
  <translation origin="center" />
  <animation pingpong="no" loop="yes" speed="150" />
</sprite>
<sprite name="walk_up">
  <image fileseq="idle/walk_up.png" leading_zeroes="3" />
  <translation origin="center" />
  <animation pingpong="no" loop="yes" speed="150" />
</sprite>
</resources>

```

The above visual profile is for a sample player character and defines four different states, though typically such a file would define many others. These states are: `idle_left`, `idle_right`, `walk_right`, and `walk_up`. The first two states are “idle” states in which the player is standing still. The two walk states are for walking right and for walking up (for a game with a top-down view). Let’s take a closer look at the profile states and sprite entities of the visual profile XML file.

```
<anim state="idle_left" spritename="idle_left" mirrorx="no" />
```

- Anim state nodes define a single entity state. Each anim states features a name tag to specify the name of the state; a sprite name that corresponds to a sprite node specified further down in the XML file, which is a reference to the image resource to use for the entity when in this state; and a mirror tag, which can be either “yes” or “no” and determines whether the associated visual style (as specified by sprite name) will be flipped or reversed. This last property is useful for directional states such as `walk_left` and `walk_right`, where the same image can be flipped to face the appropriate direction for each state.


```
<sprite name="walk_up">
  <image fileseq="idle/walk_up_.png" leading_zeroes="3" />
  <translation origin="center" />
  <animation pingpong="no" loop="yes" speed="150" />
</sprite>
```

- Sprite XML nodes define a single visual style, whether an image or animation, independently of any state node, and so a single sprite node may be referenced by more than one state. A sprite node has a name (e.g., “walk_up”), and this name should correspond to the spritename node of any state that references this sprite. The image fileseq tag defines the template form, or structure, of a sequence of image file names that together constitute a complete animation for this visual style. For example, <image fileseq="walk/walk_up_.png" leading_zeroes="3" /> specifies a sequence of files in the form walk_up_000.png, walk_up_001.png, walk_up_002.png, and so on, for however many files by this name are found in the subfolder of the Worlds folder. The translation origin determines the origin from which transformations (such as moving the image) are measured; this example specifies the center of the image as the origin. Finally, the animation pingpong node specifies the speed at which the animation is to play (milliseconds per frame), and loop defines whether the animation should loop after a single cycle of frames has completed. If loop is set to true, then the pingpong tag determines the nature of the loop; “no” means the animation plays again from the beginning, and “yes” means the animation is played (or pingponged) forward, then backward, then forward again, and so on.

8.11.4 Moving a Character Using the Keyboard

A map entity such as the player or an NPC has a visual profile attached, and this profile handles the entity’s appearance as it enters different states (walking, jump, running, etc.). In addition to a change in appearance, entities like the player character should be controllable; that is, users should be able to move them to different positions around the map using the keyboard or mouse. Following is a Lua script that responds to keyboard input (such as an up arrow keypress) and moves the player to a new X,Y position appropriately.

```
RunScript("system/player_utils.lua");

//-----

function OnInit()
    this:GetBrainManager():Add("StandardBase","");
    this:SetIsCreature(true);
end

//-----

function OnPostInit()
    this:SetTurnSpeed(6);

    //Create an event link between keypresses and functions in
    //script.
    //On each keypress for directional arrows, corresponding function
    //is called
    GetInputManager:AddBinding("left,always", "OnLeft",
        this:GetID());
    GetInputManager:AddBinding("right,always", "OnRight",
        this:GetID());
    GetInputManager:AddBinding("up,always", "OnUp", this:GetID());
    GetInputManager:AddBinding("down,always", "OnDown",
        this:GetID());

    ResetKeys(); //Reset keys back to original states
    this:SetRunUpdateEveryFrame(true);
end

//-----

function OnKill()
    RemoveActivePlayerIfNeeded(this);
end

//-----

function Update(step)

    //Make camera follow the player entity as it moves around
    //the map
```

```
AssignPlayerToCameraIfNeeded(this);
local facing = ConvertKeysToFacing(m_bLeft, m_bRight, m_bUp,
    m_bDown);

//Determine where the entity is facing and set state from the
//visual profile
if (facing != C_FACING_NONE) then
    this:SetFacingTarget(facing);
    this:GetBrainManager():SetStateByName("Walk");
else
    this:GetBrainManager():SetStateByName("Idle");
end
end

//-----

function ResetKeys()

    m_bLeft = false;
    m_bRight = false;
    m_bUp = false;
    m_bDown = false;
end

//-----

function OnLeft(bKeyDown)
    m_bLeft = bKeyDown;
    return true; //continue to process key callbacks for this
                //keystroke
end

//-----

function OnRight(bKeyDown)
    m_bRight = bKeyDown;
    return true;
end

//-----

function OnUp(bKeyDown)
    m_bUp = bKeyDown;
    return true;
```

```

end

//-----

function OnDown(bKeyDown)
    m_bDown = bKeyDown;
    return true;
end

//-----

```

The above script features a series of event handlers — OnUp, OnDown, etc. — called each time the corresponding key is pressed by the player, and each handler sets a Boolean variable to true or false depending on whether the key is pressed or released. The update function is run on each frame (iteration) of the game loop, and the value of each Boolean (each indicating which key is pressed), in combination with character speed, determines the distance and the direction along which the player entity travels.

8.11.5 Clever Navigation with Pathfinding

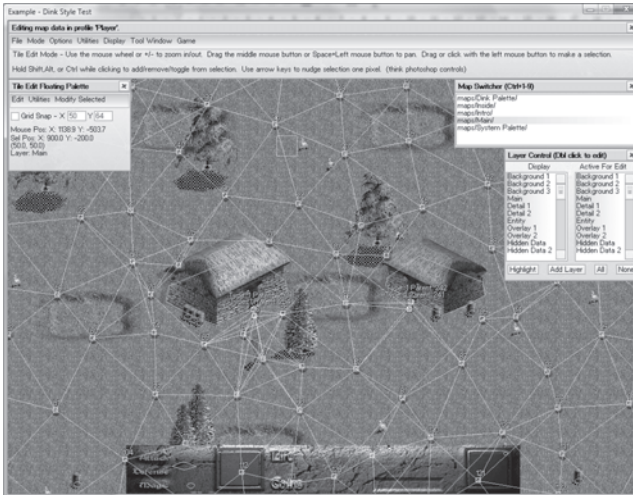


Figure 8-21:
Pathfinding nodes

The artificial intelligence term “pathfinding” refers to a computer’s ability to find, or calculate, a valid route from point X to point Y across a given graph of connected nodes. More practically, good pathfinding ensures NPCs such as enemies or other creatures intelligently find their way from X to Y across a game map without getting lost or stuck, or without bumping into obstacles or walking through solid objects. Novashell features an integrated pathfinding system that automatically performs all the required calculations necessary for NPCs to find valid routes between any two points around the map, from the point at which an NPC begins traveling through to its destination. The Novashell developer needs only to plot the points (or nodes) onto the map, forming a graph network across which pathfinding calculations are performed. In short, developers use the Novashell editor to plot nodes economically around the map (invisible to the player) such that each node is connected by a line to *at least* one other node in the network, and such that at least one node is directly reachable by an NPC regardless of where in the level he is standing. Directly reachable means there is no obstacle (like a table, wall, chair, etc.) breaking the invisible line between the NPC and the node. In Novashell, pathfinding networks are created by copying the node tile from the System Palette and pasting new instances in various locations on the level. Each newly plotted node will connect to others on the map automatically, provided a direct line of sight between the new node and the existing nodes is available. The following step-by-step process creates a path node network using the Novashell editor.

1. Beginning from the Novashell map editor, select the System Palette.
2. Select the **Path Finding Node** tile and copy it to the clipboard.
3. Select a different game map, and from the main menu select **Display | Show Path Finding Data** to display any existing path nodes and their connections.
4. Paste the path nodes around the map in a style similar to those in Figure 8-21.

In addition to pathfinding nodes, the System Palette also offers waypoints; waypoints are to path nodes what entities are to tiles. A *waypoint* is an advanced path node that can be connected to the node

network, but additionally each waypoint can be named uniquely in order to distinguish one waypoint from other waypoints and from among the mass of other nameless path nodes. Naming waypoints is useful for identifying a specific waypoint so that Lua scripts can, if required, instruct specific entities to travel to specific waypoints on a map, rather than to travel to an anonymous X,Y position. This is demonstrated in the following script.

```

RunScript("system/player_utils.lua");

//-----

function OnInit()
    this:GetBrainManager():Add("StandardBase","");
    this:SetIsCreature(true);
end

//-----

function OnPostInit()
//Add a random waypoint by name to the queue of places to which
//the entity should travel
    AddPaths();
end

//-----

function OnKill() //run when removed
    RemoveActivePlayerIfNeeded(this);
end

//-----

function AddPaths()

//Build waypoint name string and append a random number
local pointName = "Waypoint" .. random(1, 4);

//Add instruction to queue for this entity; travel to specified
//position
this:GetGoalManager():AddApproach(GetEntityByName(pointName):
GetID(), C_DISTANCE_CLOSE);

```

```
//Add instruction to end of queue (after travel) to repeat
//travel process
this:GoalManager():AddRunScriptString("AddPaths()");

end

//-----
```

8.12 Conclusion

In this chapter we examined Novashell, a free, open-source 2D game maker made by Seth Robinson for developing cross-platform games on Windows, Linux, and Mac. Like many GDKs (game development kits), Novashell is a work in progress that is continually changing. For this reason, Novashell is likely to expand and absorb a growing collection of features and quirky innovations for making games, courtesy of a fledgling community of developers and gamers. Appendix H at the end of this book features a complete list of the Novashell functions available for scripting with Novashell tiles and entities.

Chapter 9

Director and Web Games

This book thus far has considered the development of primarily two styles of cross-platform games. One type includes games written and cross-compiled in Code::Blocks C++ to run natively on each target platform (Linux, Mac, Windows, etc.). These games are usually created in conjunction with an open-source game SDK such as SDL (Simple DirectMedia Layer), which is designed specifically for drawing graphics to the game window and for playing sounds and music to the speakers to add atmosphere and realism. Games made in this style include *The Battle for Wesnoth*, *Dirk Dashing*, and *Open Arena*. The other type includes games created in a GUI game editor (such as *Novashell*), where developers build a game world by designing maps and referencing externally sourced Lua scripts, after which both the design and scripts are compiled to run as a completed game running atop a cross-platform engine. We refer to these styles of cross-platform games as “native-compiled” and “engine-based.”

This chapter considers a third type of cross-platform game especially popular on Windows and Mac known as the “web game” (or virtual machine-based game). A thriving industry and played by gamers across the world, web games are so called because they typically run through a web browser and are *embedded* inside a web page, available to most (but not all) gamers with Internet access. Some of the more well-known web games include *Diner Dash*, *Home Run*, *Samorost*, and *Bejeweled*. Often, web games are the kind people play casually to fill a rainy afternoon at home, or the kind parents show to their kids to keep them quiet for an hour or two, or the kind people

play at work to occupy themselves during a vacuous moment or to help clear their minds.

More technically, though, unlike engine-based games (such as those made with Novashell), web games are said to run in a web page through a *virtual machine* (VM). That is, developers typically create their web games by first using a GUI editor (or content creator) specifically designed for creating web games in place of an IDE like Code::Blocks. These editors usually offer tailored scripting facilities, level designing tools, sound playback features, and more. Once game development is finished and the completed game is ready to play, the whole project can be exported from the GUI editor to a *platform-independent* image file (object file). The game file then can be loaded into a virtual machine, in which the file is interpreted and played on-the-fly for the target platform, whether it is run stand-alone by an independent VM or in a web page by a browser-embedded VM.

Thus, the cross-platform compatibility of any given web game depends not so much on the specificity of the game image file created by the developers as it does on the availability of cross-platform VMs capable of playing the games. Put simply, the game image file is platform independent insofar as it may be interpreted and run on any platform for which there is a compliant virtual machine. Hence, the platforms supported by any web game depends entirely on the platforms supported by the VM; a VM for Windows means the web games play on Windows, a VM for Mac means the web games play on Mac, and so on. Overall, then, the process of running a web game (whether stand-alone or in a browser) occurs between two distinct entities: the game image itself as created by the game developers, and the virtual machine (the player of the game image, the infrastructure upon which it executes). In short, the relationship between game and VM parallels the relationship between software and hardware and between MP3 files and the MP3 player upon which the files play.

Mirroring this distinction between VM and game, there is a distinction between the VM software itself and the IDE available to create the web game; one makes the game and the other plays it. In the contemporary world of web game development there are two dominant VMs and two complementary game editors for developing web games that run in each VM, respectively (all now made by Adobe Software). These VM and editor pairs are as follows:

- **Flash and Flash Player** — Flash, originally released as a vector graphics solution by Macromedia Software in 1996, is now owned by Adobe. Flash is the name of both the GUI editor for making web games and the VM (Flash Player) that runs in a web browser and plays the games as web-embedded applications. Flash is often recognized by web developers as being a simple-to-use “light-weight” solution for creating compact animated multimedia content ranging from web games to content-rich web sites. The Flash editor is widely used by web developers, and the Flash Player supports many platforms (from Windows to Linux and Mac).
- **Director and Shockwave** — Director is a commercial (not free or open-source) “content creator” for making web games and other multimedia presentations that run both in a web browser (through a Shockwave browser plug-in) and stand-alone (where the Shockwave VM and the game are together compiled into a single executable). In short, Shockwave games can run in a web page or as stand-alone executables. The rest of this chapter covers Director in the context of creating cross-platform games.

9.1 Director

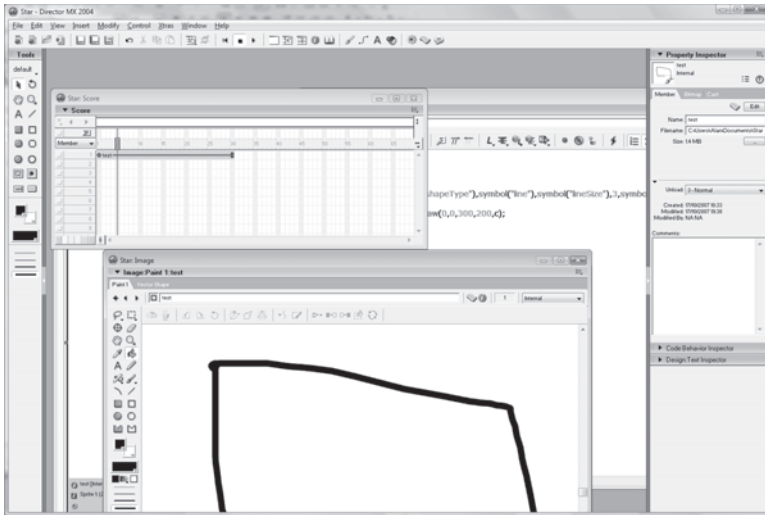


Figure 9-1

To summarize, Adobe Director (formerly known as Macromedia Director) is a GUI-based “content creator” and a complement to Adobe Flash for making cross-platform multimedia products (though Flash is not required to run Director or the Shockwave virtual machine). Director as an IDE likens a multimedia application to a play in a theatre or to a movie in which the developer takes the role of the director, or the orchestrator. Director is so called because it is based largely on a movie metaphor: one in which the developer (the director) is the manager responsible for synthesizing the ingredients of the production (the actors, the props, the special effects crew) into a coherent whole, a harmony whereby the completed product (the game, or the movie) becomes greater than the sum of its parts on account of their synthesis. So overall, Director is a complete suite of tools integrated into one application designed specifically for making multimedia productions featuring frame-based animation, vector graphics editing, media import (from files such as .mp3, .mov, .avi, .mpg, and .bmp), and also scripting and event-based programming using either the widely known language JavaScript or the Director-specific proprietary language Lingo. The latest version of Director (at the time of writing) is Director MX 2004. Although this is a commercial product, a free trial (30-day) version is

available and can be downloaded directly from Adobe's web site at <http://www.adobe.com/>.

9.2 Director Games

Adobe Director has had a long and varied history from its early beginning as Videoworks, through various name changes and upgrades to its present incarnation. Because of its long history, many developers have specifically chosen Director to create their games, and their Director games have in turn become successful. Thus, after a cursory glance of the games available on the contemporary market, it is not hard to find games created with Adobe Director. Some examples of available (or coming soon) Director games are featured below:

- **Barrow Hill** — Released by Shadow Tor Studios in 2006, Barrow Hill is a Director-made first-person adventure game (shaped in the classic style similar to Zork, Myst, or Shivers). Barrow Hill has a spooky-horror theme in which the player finds himself embroiled in a ghostly mystery after becoming stranded at an abandoned gas station in the English town of Barrow Hill after his vehicle inexplicably breaks down on the road nearby. More information about Barrow Hill can be found at the Shadow Tor web site at <http://www.shadowtorstudios.co.uk/>.

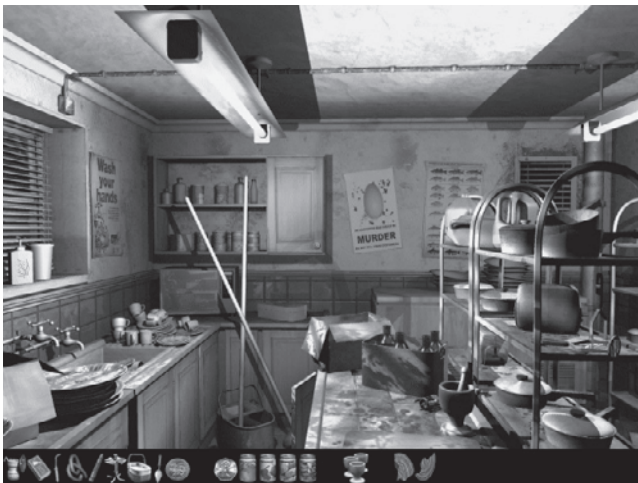


Figure 9-2: Barrow Hill: Curse of the Ancient Circle, Shadow Tor Studios, 2006

- **Diner Dash** — Diner Dash was originally released in 2004 as a Director-made web game by New York developer Gamelab, and it has since become one of the most popular and therefore most famous casual games, now having spawned a number of sequels and having been ported both to the mobile phone and console platforms. In Diner Dash, the game occurs inside a restaurant where the gamer must take the role of a waitress, Flo. Here, in a frenzied rush, she must serve the impatient diners who come to the restaurant for a meal and to socialize. In each level, Flo must seat the increasingly demanding customers at their tables, take their orders, deliver to them both their meals and their bills, and then clean up their messy tables in order to seat the next wave of demanding customers, and the mouse-clicking mania goes on. Diner Dash can be played online for free in a browser at PlayFirst, located at <http://www.playfirst.com/game/dinerdash> (requires the free Shockwave VM).



NOTE. The Shockwave plug-in for most popular web browsers can be downloaded from <http://www.adobe.com>.

- **The 13th Doll** — The 13th Doll is the unofficial sequel to the 7th Guest and the 11th Hour puzzle games released in the mid-1990s by Trilobyte. More accurately, it is a Director-made prequel to 11th Hour and a sequel to 7th Guest, made by a small independent team of volunteers under the collective banner of Attic Door Productions. In this puzzle game, the gamer may play either a doctor or his patient, whose lives cross as they enter a strange and abandoned mansion once owned by a sinister toymaker named Henry Stauf. Here, the player must move from room to room, finding clues and solving an intricate network of brainteasers to unlock the secrets of Stauf's mansion. The 13th Doll is currently in production and is planned for a cross-platform release (Windows, Linux, and Mac). More information can be found at the 13th Doll web site <http://www.t7g3.com/>.
-



Figure 9-3: The 13th Doll

9.3 Director and Shockwave Compatibility

Many of the developmental software and tools examined in previous chapters of this book (Code::Blocks, GIMP, Blender, Audacity, etc.) were cross-platform in the sense that each application supported Windows, Mac, and Linux; this meant that developers could develop games on any or all of those platforms as well as cross-compile their games to run on each of them, supporting all users on each platform natively. However, both Director (the game editor) and Shockwave (the virtual machine that runs Director-made games) sport a cross-platform “compatibility” that is based primarily on profitability and is therefore more limited and less encompassing than the cross-platform compatibility of the other development software hitherto considered. Specifically, both Director and Shockwave support only Windows and Mac *natively*; neither officially supports Linux as a platform. Thus, since Shockwave does not support Linux and since a web game (like a Shockwave game) may run only on those platforms for which there is a compatible VM available, it seems to follow that Shockwave games cannot run on Linux, only on Windows or on Mac. Officially, this is the case for now; however, alternative strategies are available for running Shockwave games successfully on Linux. In Chapter 2 we examined Linux Ubuntu in the context of running Windows applications through emulators such as Wine and CrossOver, and through transgaming technologies like Cedega. Simply by having these installed on Linux, most Shockwave games should execute successfully, but thorough and

regular cross-platform testing during game development is suggested to ensure issues (such as incompatible media formats like .mov, or certain function calls) do not arise that “break” compatibility with the transgaming emulators on Linux.

9.4 Getting Started with Director

Adobe Director is a commercial content creator driven by a movie-making metaphor that places the game developer in the position of a director. In this capacity developers use its features to create both stand-alone and web-based cross-platform games that run through the Shockwave virtual machine (supporting Windows and Mac, and Linux through transgaming emulation). This movie metaphor has come to shape the style of the features developers work with in Director. The following steps offer a simple “getting started” guide to installing and then creating a simple application with Director. Subsequent sections of this chapter examine Director and its features in more detail.



NOTE. These steps apply to Director MX 2004 for Windows.

9.4.1 Downloading and Installing Director

1. Beginning from the desktop, navigate a web browser to the Adobe web site from which to download Director at <http://www.adobe.com> (or go directly to <http://www.adobe.com/products/director/>). From there, search the product listings for Director, and click **Download Free Trial**. Save a copy of the installer from the web site to a location on your local machine.
-

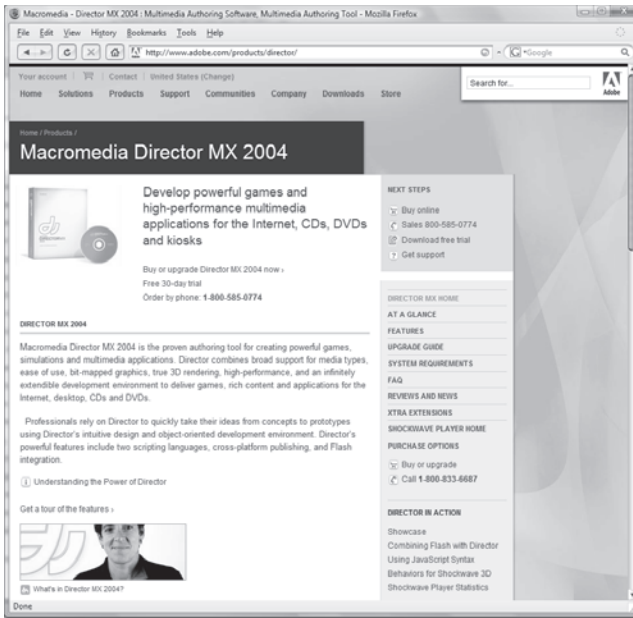


Figure 9-4: Director download site

2. Run the installer and follow the Installation Wizard to install Director to the local computer. Once completed, run Director from the Windows Start menu.

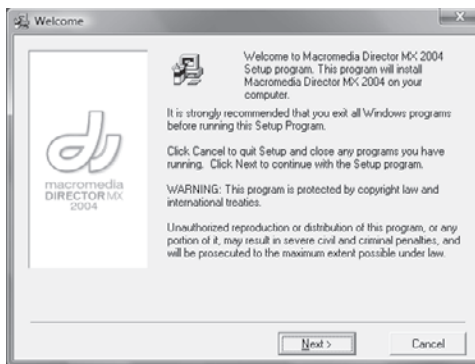


Figure 9-5: Installing Director



NOTE. Windows Vista users may need to run Director in Administrator mode.



Figure 9-6:
Configuring Director to
run on Vista

9.4.2 Creating an Animated “Hello World” Application in Director

1. Director begins at the Project Wizard (or Welcome) menu, from where developers create new projects or open previously saved projects. Click **Director File** under the Create New heading to begin a new, blank Director project.

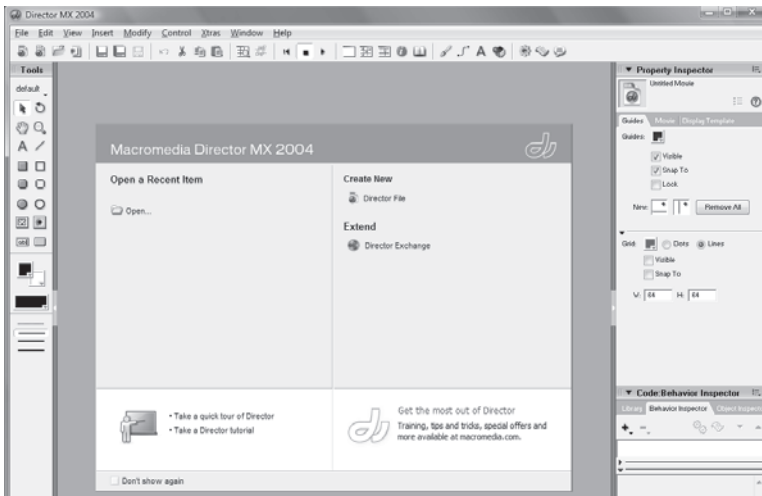


Figure 9-7:
Beginning a new
Director project

2. A blank new project is presented in the editor. This includes the Stage window, which corresponds to the game window (the output display shown to the gamer when the game is run); the Score window, graphically illustrating the game timeline that ticks over in frames per second beginning from frame 0 and moving onward incrementally, frame by frame, the moment the game is run; and the Property Inspector panel (often found docked on the right-hand side of the editor) that displays the properties (such as X Pos, Height, Visible) for the currently selected object.

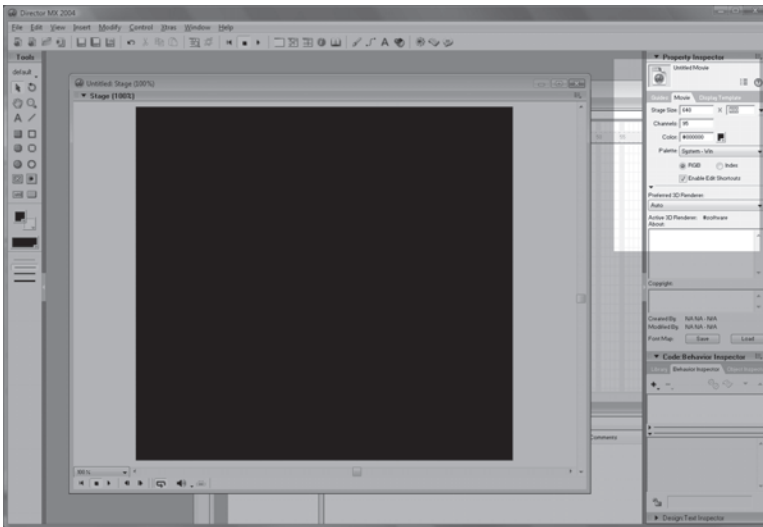


Figure 9-8: The Property Inspector panel

- Click the **Movie** tab on the Property Inspector panel and adjust the size of the Stage window (the rendered output) to 640 x 480 resolution. This changes the size of the game window.
3. The project is now ready to accept imported image data (such as BMP or JPEG files) to display to the screen in the output window. This image will then be animated to move across the Stage from the left side to the right side over the course of 30 frames. Image, movie, and sound files are collectively termed “resources” when located on disk, and resources are loaded from files on disk and into Director as “cast members.” Cast members are not added directly to the Stage, but must later be added manually.

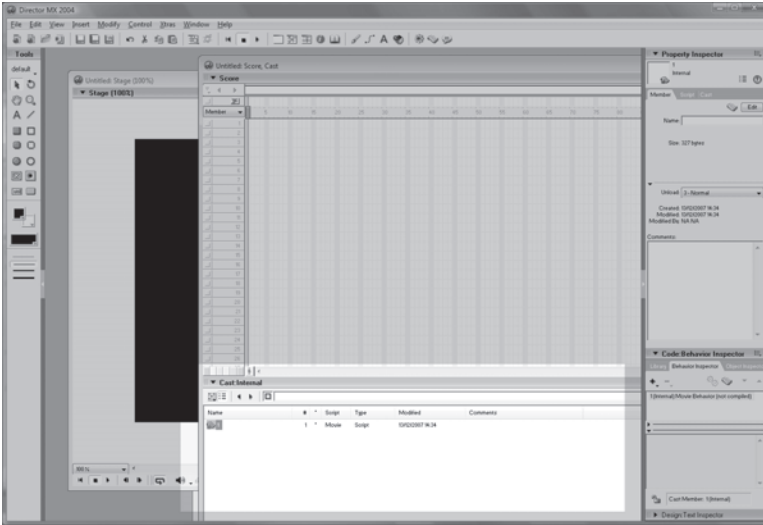


Figure 9-9: Importing image data

To import a prepared image file from disk and into Director as a cast member, activate the Time-Line window (or the Cast window) and right-click in the empty list view. When the context menu appears, click **Import** to display the Import Files dialog, from which you can select files.

4. From the Import Files dialog, select a local image file on the hard disk to import into Director as a cast member using the list view of files and folders. Once selected, activate the Media drop-down list at the bottom of the dialog, and select **Link to External File** instead of Standard Import. Standard Import instructs Director to create a duplicate file of the selected file in memory, and this is the file Director continues to reference in place of the file on disk. Link to External File means Director references the file on the disk directly. The latter option means that any changes that occur to the image on disk (perhaps a developer wishes to change the image entirely but keep the same file name) are also reflected in the image drawn on the Stage by Director at run time. Click **Import** to continue.

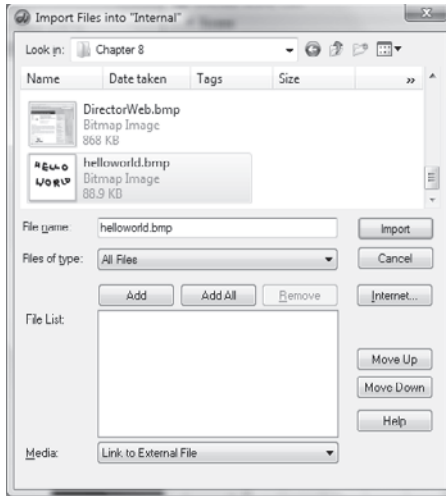


Figure 9-10: The Import Files dialog

- The Image Options dialog appears. Remove the check mark from the Trim White Space check box and click **OK** to continue. The image will be imported as a cast member for this project, and is ready to add to the Stage. The Trim White Space option deletes any white pixels in an imported image and replaces them with transparency.

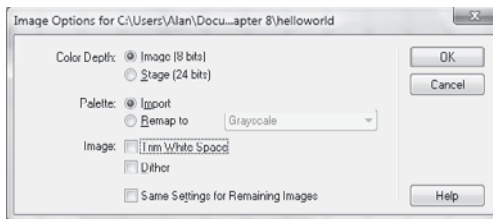
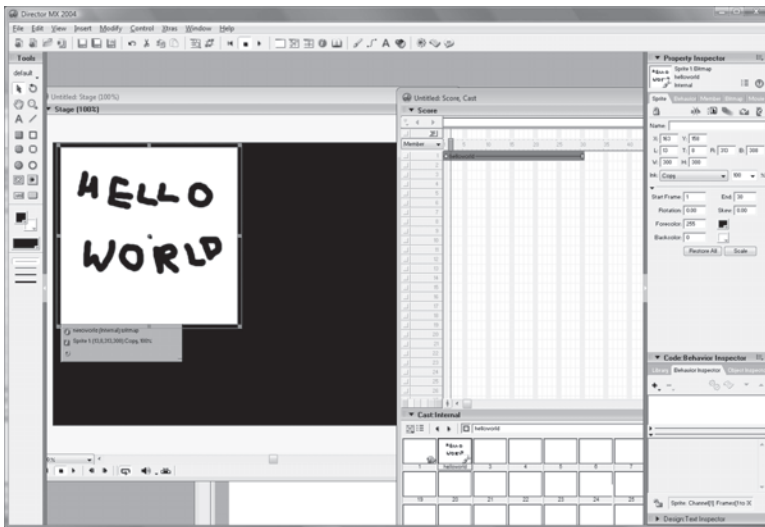


Figure 9-11: The Image Options dialog

- The image is now added to the Cast window as a cast member of the project. Cast members do not themselves appear directly on the Stage after being imported into a Director project. Rather, being a cast member means Director keeps a reference to the imported image as a potential resource for use later; either the image is referenced externally on disk (via Link to External File) or the image is loaded entirely into Director (via Standard Import). Once loaded into Director as a cast member, it can be dragged onto the Stage where it will appear as a *sprite* (an on-stage instance of that cast member), and here it will be drawn to the game window,

visible to gamers. Any single cast member may be dragged to the Stage many times, each time creating a new on-stage sprite based on the same cast member in memory, in the same way a single C++ class may be instantiated many times as different objects. Thus, many sprites may be based on the same cast member, and thereby the same resource in memory (that is, copying and pasting a sprite does not duplicate in memory the resource itself upon which it is based). For example, many on-stage tree sprites each based on the same tree cast member may be copied and pasted around the Stage to create a dense and verdant forest of tree sprites, all without duplicating their associated cast member in memory.



*Figure 9-12:
Dragging the cast member onto the Stage*

7. Once the cast member is dragged onto the Stage as a sprite, the sprite on the Stage and the Score window can together be used to animate the sprite to, for example, move it from one corner of the Stage to the diagonally opposite corner over the course of 30 frames. To do this, first activate the Score window and drag the timeline slider across the header of the timeline to frame 30; this sets the current frame to edit, namely frame 30. Afterward, activate the Stage window and click to select the sprite on-stage. In the center of the selected sprite appears a small red marker, and this can be dragged to the destination point at the corner of the

Stage where the sprite is to reach by frame 30. This then marks the path (or vector) along which the sprite is to travel, from its start point (where the sprite is situated currently) along an invisible line across the Stage to a destination point marked by the red marker.

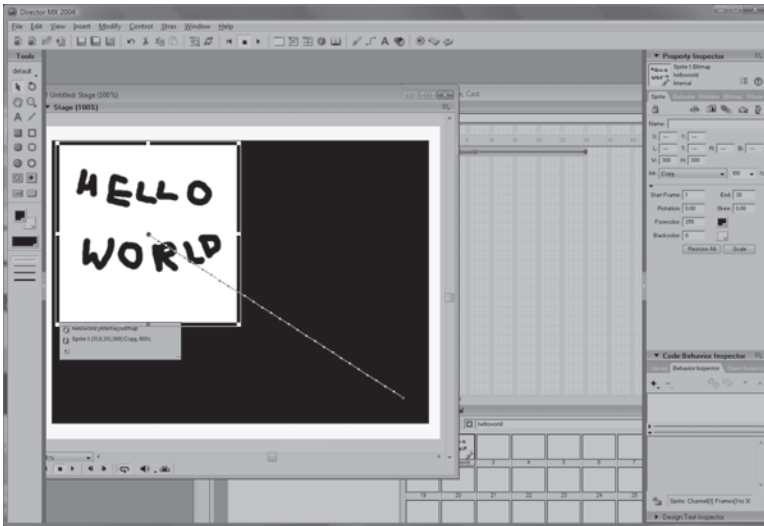
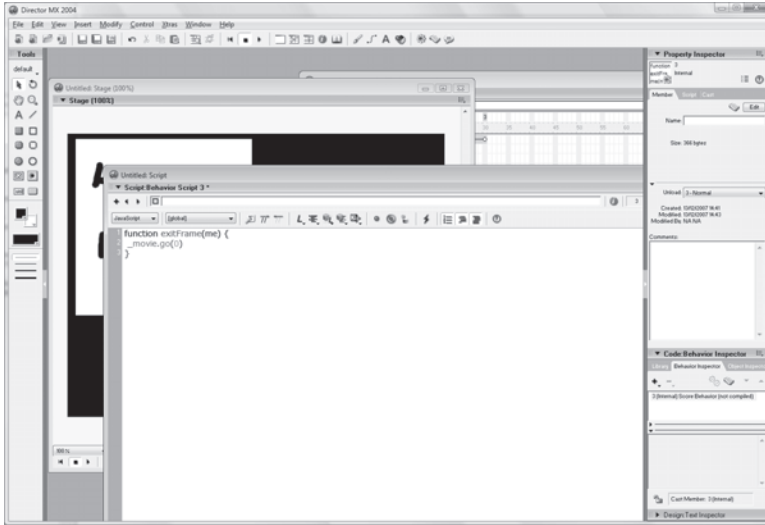


Figure 9-13:
Setting the travel
path

- The Director movie is almost ready to play insofar as a sprite on-stage will now slide gently across the Stage from a source to a destination point within 30 frames. However, once the 30 frames expire, the movie will end and the presentation will stop. To configure the movie to run continuously until the player presses Stop, scripting is required. Double-click the blank cell above the 30-frame header in the Score window to open the Scripting window, where the developer can edit the script for the current frame. Select the scripting language JavaScript from the drop-down menu, and then enter the code shown below into the code editor. This code instructs the Director movie to shift playback to the first frame of the movie whenever the timeline reaches the end of frame 30.

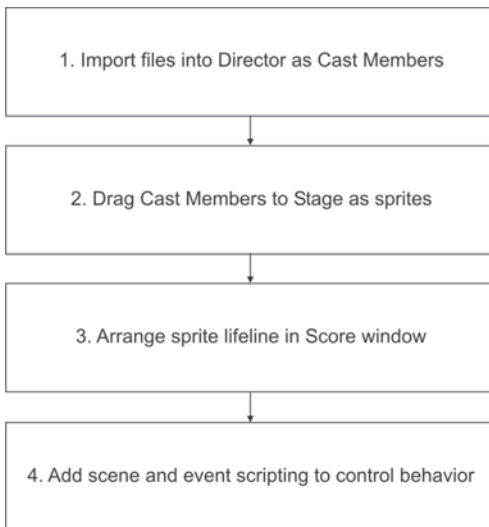
```
function exitFrame(me)
{
    _movie.go(0);
}
```



*Figure 9-14:
Adding JavaScript
code to control
playback*

9. The “Hello World” Director movie is now completed. Click the **Play** button on the toolbar to begin movie playback.

9.5 Director in More Detail



*Figure 9-15:
Director workflow*

In summary, Adobe Director is a commercial content creator whose features are designed specifically for developers making cross-platform games and other interactive presentations, from animated menus to real-time 3D games. As an IDE (integrated development environment), Director conceptualizes game development as being akin to a movie production and offers to developers several core components. Let's look at the components in detail.

9.5.1 Cast Members

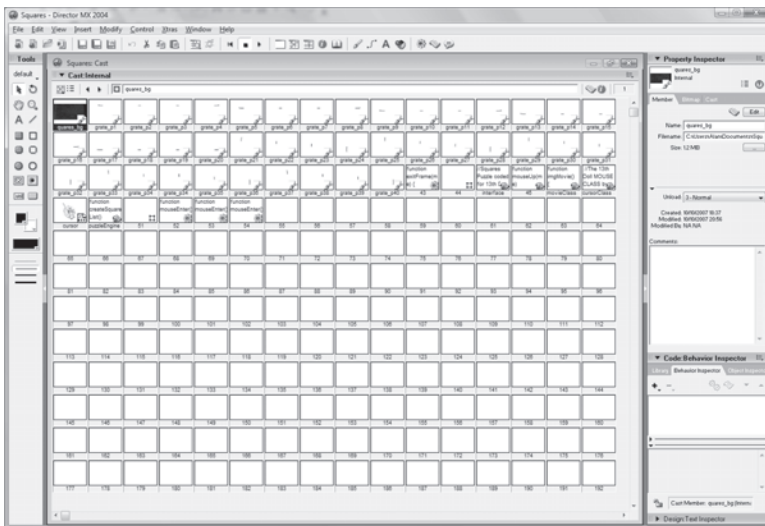


Figure 9-16: The Cast window containing cast members

Games are said to be greater than the sum of their constituent parts in the same way a movie is more than just the sum of its cast members. Games are more than just a soulless list of graphics, sounds, and music. Rather, the game emerges as a feature-filled totality from the particularity of its recipe, from the unique blend and configuration of its resources or cast members. Thus, in the context of games made in Director, a cast member refers to a resource (an image, a sound, a movie, a text file, or any other externally loaded file brought into the game), and the game as a whole is then assembled step by step by piecing together the cast members into an original formation on the Stage. Thus, the Director Cast window is a data bank of resources; a collection of files imported into Director as cast members.

More accurately, some cast members may be imported and some are instead linked. Imported cast members are loaded into Director as static, byte-for-byte copies in memory, and they persist unchanged regardless of what changes may occur subsequently to the contents of the file from which they were loaded. The imported cast members are independent of the resource files. For example, a bitmap of a bright yellow happy face may be imported into Director as an image cast member. As such, Director makes a copy of the imported file and it is this copy that Director continues to reference, not the original file. Hence, the imported clone remains unchanged even when the original file is deleted or changed. Linked cast members, by contrast, are linked to the external files from which they are loaded into Director; thus, changes in the file are reflected in the cast member.

Cast members have a variety of properties that can be changed using the Property Inspector panel. Two of these properties include:

- **Name** — The name is a programmer-defined string to be used for the cast member. This property is optional but useful for scripting, as we shall see later.
- **Number** — Each cast member in any single Director project is assigned a number automatically, and every number must be unique to a single cast member. This property can, however, be changed manually, usually for scripting purposes.

9.5.2 The Stage

The Stage corresponds to the game window; that is, the final viewing area for the gamer. To continue the movie analogy, to be on-stage is to be visible and to be “off-stage” is to be unseen, to be in the wings. Overall then, the Stage is a place for actors or members of the cast; thus, the Cast window and the Stage window are intimately related, in the same way a catalogue is related to the objects that are catalogued. Cast members are dragged from the Cast window and into the Stage window where they appear on-stage as sprites, ready to perform. The relationship between a cast member and a sprite can be either one-to-one or one-to-many, where a single cast member may be dragged to the Stage once or many times to create one or many sprites. Each sprite is based on the same cast member, just like a single C++ class declaration may give birth to many instances of that type, or a single

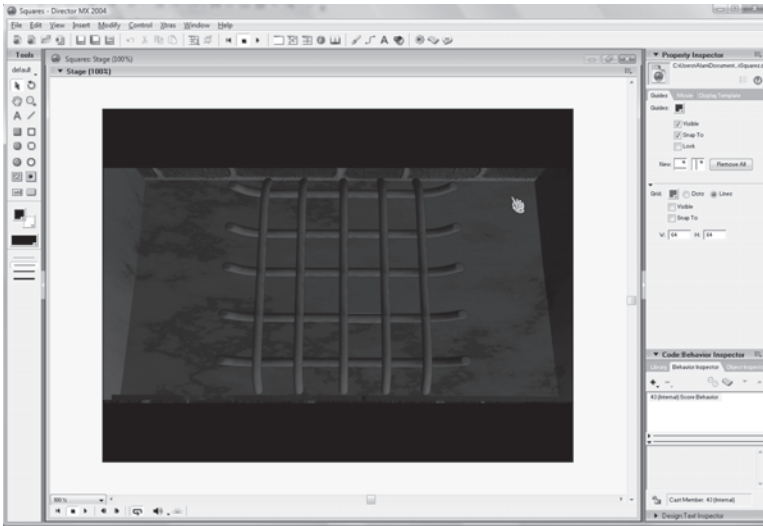


Figure 9-17: A file displayed on the Stage

word-processor letter template may give rise to many letters based on that single template. Here, on the Stage, sprites are positionally arranged by the developer by clicking and dragging them from the Cast window and into the Stage or the Time-Line window. Most programming in Director will occur in relation to sprites, such as defining where and how sprites move on-stage, determining when sprites collide with each other (collision detection), determining how sprites are z-ordered (arranged to the display), and so on.

The Stage has many properties that can be edited using the Object Inspector panel. These include the following:

- **Stage Size** — The width and height of the Stage specified in pixels; changing these properties adjusts the screen resolution (or window size) at which the game is run.
- **Title** — A programmer-defined string specifying the title of the game window to be used while the game is running.
- **Color** — The 32-bit color value specifying the color to be used for the Stage background; set by default to black.

9.5.3 The Score Window's Timeline

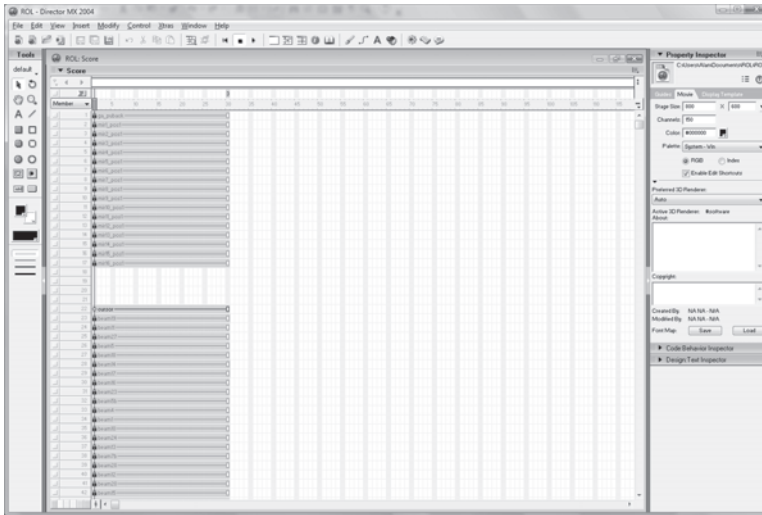


Figure 9-18: The timeline is shown in the Score window.

The Time-Line window displays a chronological map of the entire Director project, documenting the comings and goings of sprites on-stage across the duration of the presentation. It details the points when sprites enter the Stage, the moment sprites leave, and the locations and transformations of sprites in any one frame. At the moment playback begins by pressing the Play button from the toolbar, a time slider runs horizontally across the header-axis of the map, ticking over the linear progress of the presentation as it unfolds frame by frame from beginning to end, left to right, just like the progress slider of a media player. During movie playback the time slider moves horizontally (from left to right) across the frames of the Score window, stepping across each sprite in its associated channel. These channels are arranged row by row, with one channel corresponding to one row.

A Director presentation consists of potentially many channels, and each channel serves an important purpose for the presentation. First, sprites find their way onto the Stage by occupying a channel. Each sprite *must* occupy a channel if it is to exist on the Stage, and each channel may accommodate *only* one sprite in any one frame. A channel may be empty if there are more channels than sprites. This means there may be as many sprites on-stage at any one time as there are channels available. Second, channels by default represent the z-order

of sprites; that is, the order in which sprites are drawn to the Stage. Specifically, sprites in channels are drawn to the Stage in order of the channels; that is, downward according to the vertical arrangement of channels in the timeline, from top to bottom, row by row, channel by channel. This means sprites in high-order channels are drawn atop those in lower-order channels. Thus, channel 0 is the background, and the highest channel represents the sprite closest to the camera (or audience).



TIP. Take a look at the Time-Line window and the presentation of channels by dragging a cast member to the Stage as a sprite, and then examine the sprite's channel occupancy in the timeline. Channels that are occupied by sprites are shaded with a colored block instead of appearing blank, and the length of the block can be resized to determine the lifespan of the sprite on-stage. That is, the sprite remains on-stage only so long as the length of its corresponding block (located in one of the timeline channels) remains *beneath* the time slider, which moves linearly across the header of the chart as the presentation is played back frame by frame. Blocks can be resized and dragged throughout the timeline.

9.6 Director Scripting with JavaScript

Director offers to developers a comprehensive scripting system integrated into the editor in order to customize and control games programmatically. Through scripting, developers can program almost every aspect of a presentation — from the run-time positioning and transformation of sprites on-stage to the ordering of channels and playing of sounds and music. Director as an IDE offers two different scripting languages for creating scripts: the Director-specific language Lingo, and the more widely known JavaScript. In this chapter, though, we concentrate on scripting with only JavaScript. Why JavaScript and not Lingo? Partly because JavaScript as a language is general-purpose and widely used by developers both inside and outside Adobe Director, and partly because JavaScript is supplemented by mammoth documentation and tutorials both online and in books, all available at the touch of a button.

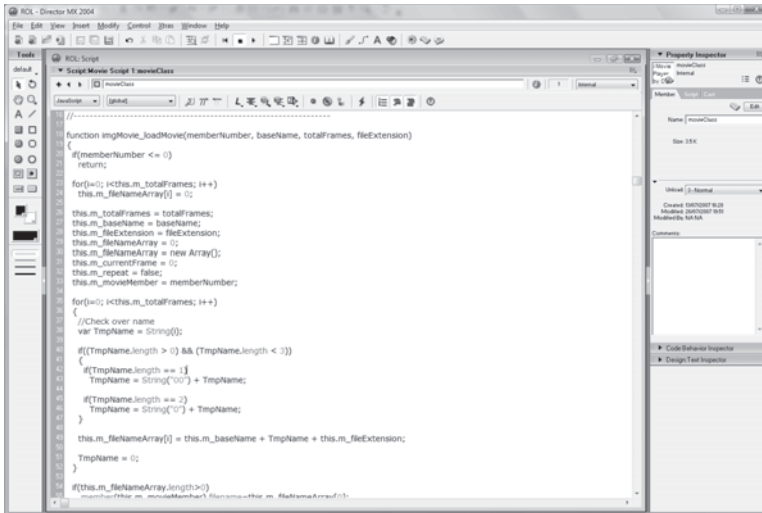


Figure 9-19: A global event script

In Director there are primarily three types of scripts, each distinguished by the way in which they are invoked; that is, the agent or event that causes the script to run. The three types are frame scripts, global event scripts, and local event scripts; these are now considered more closely.

9.6.1 Frame Scripts

Frame scripts are specifically timed scripts. They occur on a specified occasion in the movie during playback or after a specified delay as measured in frames. Frame scripts are created via the Time-Line window by clicking on any of the vacant cells above the time slider header region to open a scripting widow. Developers enter script into the editor that is to execute on the selected frame. In short, frame scripts are attached to specified frames in the presentation, and the script executes whenever the time slider enters that frame during playback. An example was the “frame loop” script created for the sample “Hello World” application in Section 9.4.2. Here, this script executed whenever the slider reached frame 30, and it sent the slider back to frame 0 where it again repeated the presentation from the beginning.

To create frame-based scripts:

1. Activate the Time-Line window in the Director editor to view the movie chronological map, a view charting the progress of the movie from beginning to end, left to right.
2. Locate the time slider in the horizontal margin at the top of the timeline, and double-click inside an empty cell above the time slider.
3. The frame script for this frame is now ready to code in JavaScript using the script editor.

9.6.2 Global Event Scripts

Global event scripts are specifically named JavaScript functions that Director searches for and runs (if present) whenever associated global events occur during movie playback. Put simply, global events are scripts run whenever an important event occurs during playback. For example, Director searches all scripted functions for one named `startMovie` as soon as movie playback begins; if found, the function is run. Likewise, Director searches for a function named `stepFrame` to execute once every frame, as each frame passes during playback. (In some senses, the `stepFrame` global event is analogous to the `Update` function featured in the game loop used earlier in this book for SDL applications.)

To create global event scripts:

1. Open the Script window by clicking the **Script Window** button on the Director toolbar. Create a new script file by clicking the **+** button.
2. Create a specifically named function as a handler for a global event. These functions must have appropriate names. The global event handlers are listed in Appendix I, “Director Events,” and can also be found in the official Director documentation. Here is an example function:

```
function startMovie()
{
    //Insert code here
}
```

3. Assign the script global scope so Director can locate the event handler in memory. To do this, activate the Property Inspector panel and under the Script tab, select **Movie** from the type drop-down box to assign the script a movie scope; that is, global scope.

9.6.3 Local Event Scripts

Local event scripts are micro-level (sprite-level) scripts attached to specific sprites on-stage in order to handle incoming events for that sprite. They handle mouse click events, keypress events, mouse enter events, and all kinds of other events that may occur to on-stage sprites during movie playback.

To create local event scripts:

1. In the Director editor, select an on-stage sprite for which to create a scripted event handler.
2. Right-click the sprite to display a context menu, and select **Script** from the menu to display the script editor window.
3. Create a script for an appropriate event handler. Sprite events are listed in Appendix I, “Director Events.” Here is an example handler:

```
function mouseDown()  
{  
    //Insert code here  
}
```

9.7 Practical Scripting

Scripting in Director with JavaScript offers developers the ability to customize at run time most elements of a Director movie, from the orientation of a sprite on-stage to the background color of the Stage itself. Animations, then, are created by interpolating these properties over time, frame by frame, throughout a single movie. Generally, scripting in Director takes an object-oriented approach. Director offers a number of important classes (core objects) for dealing with elements in a movie, from sprites and movie playback to channels and cast members. Some of the common Director classes encountered by a developer while scripting are:

- **Movie** — The Movie class is the topmost controller of a movie in a hierarchy of classes with different status. This class exposes methods and properties for dynamically adding and deleting frames to and from a movie, respectively; looping from one frame back to another; and refreshing the contents of the Stage manually (though this usually occurs automatically).
- **Player** — The relationship between the Director movie and the Player class parallels that between a song playing in a media player and the media player's controls (forward, rewind, pause, etc.). In short, the Player class allows developers to control movie playback at run time.
- **Sprite** — A *sprite* is an instance of a cast member on-stage, and every Director movie contains in script an instance of class Sprite for every sprite on Stage. It is through this class that any individual sprite may be controlled. Using the Sprite class, developers can set the position, size, and orientation of sprites; change sprite channels; and hide and unhide sprites.
- **Channel** — A *channel* refers to the track or slots in the Time-Line window, and the purpose of a channel is to accommodate one sprite on-stage at any one time. Channels can be accessed and controlled programmatically through the Channel class and, like sprites, there is one instance of a Channel class in memory for every channel in the Director movie, regardless of whether or not the channel is vacant.

- **Member** — The Member class refers to a cast member (that is, a resource imported from a file on disk), and there is one instance of Member for every cast member present in the movie.

Let's look at some practical scripting tasks commonly faced by Director programmers. Many of the samples will make use of the aforementioned "core classes" considered here.

9.7.1 Programming: Shapes, Lines, and Primitives

In addition to the premade art, music, and sound imported from resource files on disk and then into Director as cast members, Director scripting further allows developers to draw pixels to the Stage programmatically in the form of dots, lines, and shapes. Consider the following JavaScript code. This code assumes a bitmap has already been imported into Director as a cast member called "test," and further that a sprite instance based on this member has been created on-stage. The sample code then draws to the on-stage sprite a line starting from one X,Y corner of the sprite and moving diagonally toward the other.

```
function startMovie()
{
    //Create array variable holding information about primitive
    //to draw

    var c = propList(symbol("shapeType"), symbol("line"),
        symbol("lineSize"), 3, symbol("color"), color(100, 0, 0));

    //Draw
    member("test").image.draw(0, 0, 300, 200, c);
}
```

This code is featured in the startMovie global event handler, called automatically by Director as movie playback begins.

The variable var c is declared as a Director-specific data type called a property list (PropList), which is an array of properties arranged sequentially in memory. A property list type is similar to the std::vector class of the STL. In this case, the property list var c is a

collection of elements describing the geometric properties of a primitive to draw.

The draw method is a member of the Bitmap class, and this in turn is a member property of the Member class (an instance of Cast Member). The draw method draws pixels to a device context, and the method takes the following form:

```
draw(x1, y1, x2, y2, colorObjOrParamList);
```

9.7.2 Printing a List of All Sprites On-stage

Resources are imported into Director as cast members, and cast members are dragged onto the Stage as sprites where they occupy available channels. For sprites to be on-stage they must occupy a channel, and only one sprite may occupy a channel at any one time; this means the number of sprites that may appear together on-stage at any one time is limited to the number of available channels in the Director project (this limit is by default 65). Thus, a Director project may be conceptualized as a collection of arrays since it contains a list of cast members imported from files, a list of channels for on-stage sprites to occupy, and a Stage, which may be considered as a meeting place of sprites. Director holds all of these elements (sprites, members, and channels) in three globally available lists, one list for each type. Consider the following code that cycles through all the sprites on-stage, and prints the name of each sprite in a pop-up message box.

```
function mouseUp(me)
{
    var TotalChannels = 65;
    for (var i=1; i <TotalChannels; i++)
    {
        if(sprite(i).Name!= "")
            _player.alert(sprite(i).Name);
    }
}
```

The global function `sprite` returns a pointer to the specified sprite, as selected by the argument index.

The `alert` function is a method of the `Player` class, and it presents to the screen a message box featuring the specified string.



NOTE. Sprites can also be selected by name rather than by number:

```
//Example to make invisible a sprite selected by name  
var tmpSprite = sprite("name");  
tmpSprite.visible=false;
```

9.7.3 Animating Sprites Using Cast Members

Each sprite on-stage occupies a channel and each is an instance of a cast member. In this respect the relationship between cast member and sprite is in the order of one-to-many; that is, any one cast member may be dragged onto the Stage one or many times to create one or many sprites. Each of those sprites is based on the same cast member in memory, and therefore each sprite appears identical to any others created from the same member. The sprites themselves may have different positions, sizes, orientations, and transparency values from others on-stage since geometric properties are in no way encoded into cast members, but the underlying appearance of each sprite (the pixel data itself) will reflect the appearance of the cast member as imported or referenced from a resource file on disk. This means that as a cast member changes (either pixel data or file reference altogether), so will all its dependent sprites on-stage, each changing to reflect the status of the cast member since it was from this cast member that each sprite was created.

The dependency, then, between cast member and sprite offers a solution for creating animated sprites on-stage. Not animated in terms of transformation, where a sprite changes position from one place on-stage to another. Nor in terms of orientation or rotation, where the geometric position of the sprite is changed. This can be achieved already through the `width`, `height`, and `locX` and `locY` properties of the `Sprite` class. But rather in terms of the sprite pixel data itself, in terms of frame-based animation, such as a man walking, an NPC punching, or an explosion sequence. Specifically, any sprite can be animated during

movie playback by changing the file reference of its underlying cast member on a frame-by-frame basis so the sprite on-stage comes to reflect the different member changes as each frame passes.

An example would be a side-scrolling platform shooter that features an NPC animation in which an NPC fires a weapon. The animation consists of ten frames, each of them illustrating the motion of the NPC as he draws the gun from its holster and finally to the point where he fires the gun, and from its barrel emerges a muzzle flash. Here, frame 1 would initially be imported into Director as a link (not standard import), and then the cast member is dragged to the Stage where it appears at frame 1. Then on each frame, the file reference of the cast member is incrementally set to the next frame so the sprite on-stage appears to cycle through the frames and thus draws the weapon and fires the gun. Consider the following code:

```
//Movie begin event
function startMovie()
{
    //Create a new global array to hold a list of file names
    //for each frame of animation
    _global.g_fileNameArray = new Array();
    _global.g_CurrentFrame = 0;

    //Add file names
    _global.g_fileNameArray[0] = "frame1.bmp";
    _global.g_fileNameArray[1] = "frame2.bmp";
    _global.g_fileNameArray[2] = "frame3.bmp";
    /...
}

//Called each frame
function stepFrame()
{
    _global.g_CurrentFrame = _global.g_CurrentFrame + 1;

    member("mytestmember").filename="@"+_global.g_fileNameArray[
        _global.g_CurrentFrame];
}
}
```

9.7.4 Querying Mouse Events

The standard `OnMouseDown`, `OnMouseUp`, `OnMouseEnter`, and `OnMouseLeave` events are sprite-based event handlers used to detect and respond to the movements of the mouse cursor over specific sprites on-stage at run time. However, developers may also wish to query the movements of the cursor for purposes more broad than this, such as to detect whether the cursor is hovering over a region of the Stage or a collection of sprites, or to track the movements of the cursor independently of sprites and wherever on the Stage it travels, frame by frame. To do this, Director offers a global instance of the `Mouse` class. Consider the following code:

```
//Determine if mouse intersects sprite("test")
if(_mouse.mouseLoc.inside(sprite("test").rect)==true)
{
    //_mouse.mouseLoc.locH; XPOS
    //_mouse.mouseLoc.locV; YPOS
    //Do stuff here
    return;
}
```

9.8 Using the Projector for Web-based and Stand-alone Games

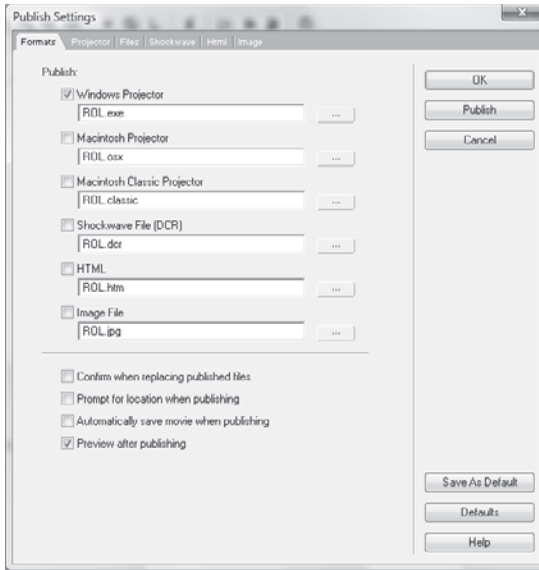


Figure 9-20

Adobe Director is primarily a GUI IDE with scripting facilities attached, and these together constitute the tools available to make web games and stand-alone games for both Windows and Mac. Until now, Director games were tested and run by pressing the Play button on the editor toolbar; press this to begin movie playback. Here developers could preview their movies/games on the Stage within the editor window. However, developers are likely to require that their completed games run independently of the Director editor — either in a web page as a Shockwave presentation or in an executable where the Shockwave player and the game image are compiled into one package. The following sections explain how to build both web games and stand-alone executables using Director.

9.8.1 Building Web Games

1. From the Director main menu, select **File | Publish Settings**.
2. Deselect all check boxes, then check the **Shockwave File (DCR)** check box and click **OK**.
3. Click the **Shockwave** tab and adjust the JPEG image quality to **80** using the slider control.

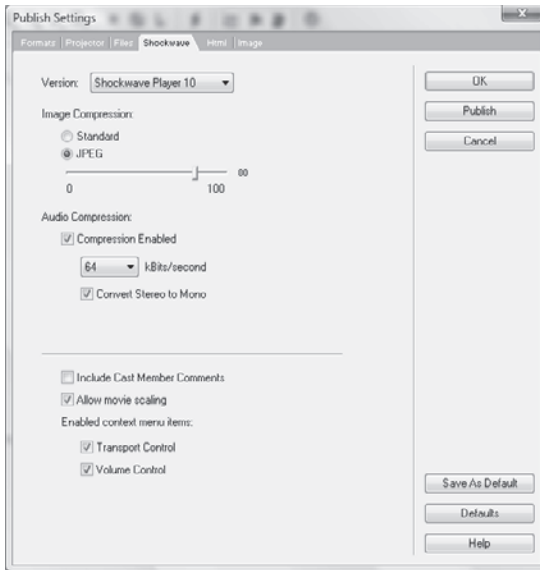


Figure 9-21

4. Click **OK**.
5. From the Director main menu, select **File | Publish**.

Once published, the project is compiled to a web game saved as a Shockwave object embedded in an HTML web page. These files are saved locally inside the project directory.

9.8.2 Building Stand-Alone Games (EXE for Windows, OSX for Mac)

1. From the Director main menu, select **File | Publish Settings**.
2. Deselect all the check boxes, then check the **Windows Projector** check box to compile for Windows, or the **Mac Projector** check box to compile for Mac.
3. Click **OK**.
4. From the Director main menu, select **File | Publish**.

Once published, the project is now compiled to a stand-alone game in the form of an executable file. This is saved locally inside the project directory.

9.9 Conclusion

In summary, Adobe Director is a commercial content creator for producing cross-platform (Windows and Mac) games officially, and unofficially for Linux through an emulator such as Wine. This chapter brings to a close the examination of 2D cross-platform games, and here this book changes direction by inspecting the world of cross-platform 3D game development using the open-source and freely available OGRE 3D API. It is this subject that is now considered.

This page intentionally left blank.

Chapter 10

3D Games with OGRE 3D

To summarize the book thus far: Chapter 1 considered the beginnings of cross-platform game development by highlighting the meaning of the term “cross-platform,” and by also examining how to run multiple platforms (Linux, Mac, and Windows) through multiple booting and virtualization. Chapter 2 examined the Linux platform generally in terms of the basics as well as some of its available features such as command line execution and the GCC compiler. Chapter 3 then considered the variety of developmental tools available for developing cross-platform games, namely the C++ IDE Code::Blocks, the photo editing suite GIMP, and the 3D rendering software Blender 3D. From here, this book considered mainly 2D games as created by the SDL (Simple DirectMedia Layer), the Novashell game editor (a cross-platform game engine), and Adobe Director, designed especially for developing web-based games played by the Shockwave virtual machine. This chapter changes focus from making 2D games to making 3D games by considering the freely available and cross-platform OGRE 3D API, compatible with the Code::Blocks C++ IDE.

Any game designated as “3D” typically falls into one of two further sub-categories, the more common real-time 3D or the less common prerendered 3D.

- **Real-time 3D** — In real-time 3D games, events can occur simultaneously (such as gun fights, explosions, and tactical ops); that is, events occur in “real time.” Games such as Doom, Unreal, Quake, Gears of War, and Carnival (on Wii), are considered to be real-time. In real-time 3D, gamers can move smoothly through the game world, rotating their perspective on demand to view game objects (walls, doors, enemies, etc.) from the front, back, underside, and potentially from as many different angles as any 3D space can physically allow. In this world of 3D, game objects are typically

composed of polygons. Books, doors, people, and more are all assemblages of hundreds, perhaps thousands, of small triangles angled and connected in specific ways. To enhance its realism, each object is then *textured*, which means its 3D surface is wrapped or wallpapered with 2D images. The 2D images can be bricks on a brick wall, wood grain projected onto the 3D surface of a wooden cabinet model, and so on.

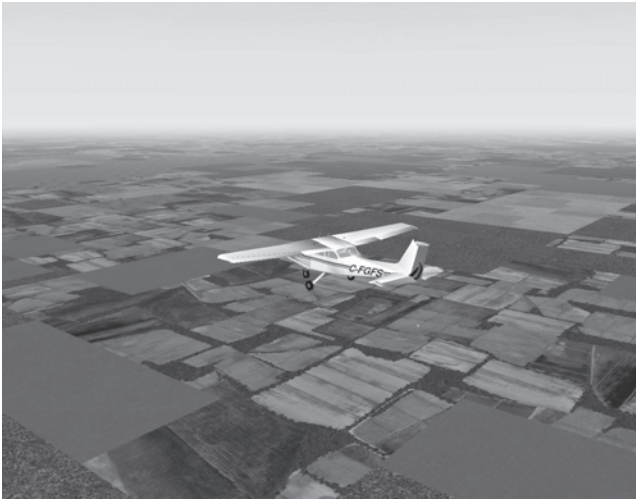


Figure 10-1: Free, cross-platform 3D flight simulator FlightGear, available at <http://www.flightgear.org/>



Figure 10-2: Free, cross-platform, and open-source 3D shooter Nexuiz, available at <http://alientrap.org/nexuiz/>

- **Prerendered 3D** — Games like *Myst*, *The 7th Guest*, *Shivers*, *Mortimer Beckett*, and *Post Mortem* are among those designated by game critics as prerendered 3D. They are prerendered insofar as objects in the game world are first modeled and rendered in 3D by artists using software such as 3ds Max or Blender 3D, and from there are imported into the game as either inflexible still images or preconfigured movies, instead of being imported as polygonal models viewable from all angles in “real time.” For example, the first-person game *The 7th Guest* finds the player wandering around the rooms of an old mansion solving brain-teasing puzzles room by room (chess puzzles, word games, etc.). Here, the player clicks the Forward button to walk forward, and a predetermined “walk forward” animation is played until the character reaches its destination, whereupon the player again may resume control and navigate to other places. Prerendered 3D is not considered further in this chapter. Instead, OGRE 3D is used here to create real-time 3D games.



Figure 10-3: The Battle for Wesnoth prerendered 3D game is free, open-source, and cross-platform; available at <http://www.wesnoth.org/>

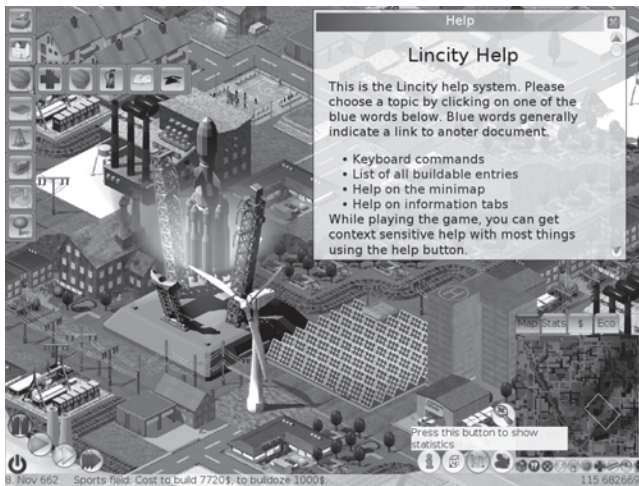


Figure 10-4: Screenshot from the prerendered game *Lincity*, which is free and cross-platform; available at <http://lincity.sourceforge.net/>

10.1 OGRE 3D



NOTE. This chapter can only be considered a gentle introduction to developing cross-platform 3D games using OGRE 3D, rather than a complete guide. This chapter covers many diverse and interesting facets of 3D games, and as such the material is condensed into lighter reading in the form of a question and answer format.

Q. So exactly what is OGRE 3D, and why would I want to use it?

A. Cross-platform, open-source, and freely available, OGRE 3D is an acronym for Object-oriented Graphics Rendering Engine, an API for drawing hardware-accelerated 3D graphics to the game window in real time (on-the-fly). In other words, OGRE 3D can make real-time 3D games. In the words of the OGRE 3D web site:

“OGRE (Object-Oriented Graphics Rendering Engine) is a scene-oriented, flexible 3D engine written in C++ designed to make it easier and more intuitive for developers to produce applications utilising hardware-accelerated 3D graphics. The class library abstracts all the details of using the underlying system libraries like Direct3D and OpenGL and provides an interface based on world objects and other intuitive classes.”

(Quoted from the OGRE 3D web site at http://www.ogre3d.org/index.php?option=com_content&task=view&id=19&Itemid=79.)



NOTE. The term “hardware acceleration” refers to the process by which software (such as games) is *accelerated* by dedicated hardware. For example, graphics cards like those made by ATI or NVidia are hardware dedicated to the task at hand.



NOTE. At the time of writing, OGRE 3D is distributed to developers as a dynamically linked library under the GNV LGPL (Lesser General Public License), the details of which are featured in Appendix A at the back of this book. More details can also be found at <http://www.ogre3d.org/>.

There are many reasons why the OGRE 3D API may be appealing to a game developer; some of these include:

- **Free, open-source, and cross-platform** — OGRE 3D is compatible with the Code::Blocks C++ IDE and GCC compiler for making 3D games, and therefore OGRE-powered games may be cross-compiled to run natively on the Linux, Mac, and Windows platforms. Furthermore, OGRE 3D is both free to download and free of charge for both commercial and non-commercial usage, as defined by the OGRE 3D open-source license available at the web address above.
- **Comprehensive documentation and thriving community** — OGRE 3D as an API is complemented by a comprehensive help file distributed together with sample applications in a complete downloadable SDK package. The OGRE web site contains comprehensive online documentation, including a community-edited wiki database. There is also a diverse and thriving online community of developers who participate in both technical and social discussion on the OGRE forums.
- **Feature-rich SDK** — The OGRE API boasts a complete structure of features specifically designed for developing real-time 3D games. These features include the following, some of which are considered later in this chapter:
 - Object-oriented scene hierarchy for programmatically creating and managing objects in real-time 3D scenes.

- Ready-to-use skeletal animation system for animating 3D models via a network of connected bones through a single bone hierarchy. (For example, to animate the arms of an NPC enemy bot, you rotate the shoulder joint and all dependent bones (forearm, etc.) are transformed relatively.)
- Post-production special effects such as ribbon trails, particle systems, bloom, motion blur, and more via the Compositor.

10.2 OGRE 3D Games

Q. Okay. I now understand more accurately the nature of OGRE — what it is and at least some reasons as to why it might be chosen by developers for creating cross-platform 3D games. But are there any working OGRE examples, any OGRE-powered games currently available to buy or to download?

A. Yes. There are many OGRE-powered games available; some free, some commercial. Some of these OGRE-powered games are featured below.

10.2.1 Ankh



Figure 10-5: Screenshot from Ankh by Deck13 Interactive

Originally developed by Deck13 Interactive in 2005, *Ankh* is a cross-platform 3D adventure game similar in style to the classic LucasArt's *Monkey Island* series, and *Ankh* itself has now spawned a sequel — *Ankh: The Heart of Osiris*. Both games were developed using OGRE 3D. In *Ankh*, the gamer controls Assril, the son of an ancient Egyptian architect living in Cairo, who is embroiled in an enigmatic quest to lift the curse placed upon him by a mummy whose sleep he once disturbed while playing inside the Great Pyramid at Giza. More details regarding *Ankh* can be found at <http://www.ankh-game.com/>.

10.2.2 Other Games

OGRE has powered a great variety of games, including *Jetracer* and *Billiards Complete*, shown in the following images.



Figure 10-6: Screenshot from OGRE-powered game *Jetracer*, by Winnerone

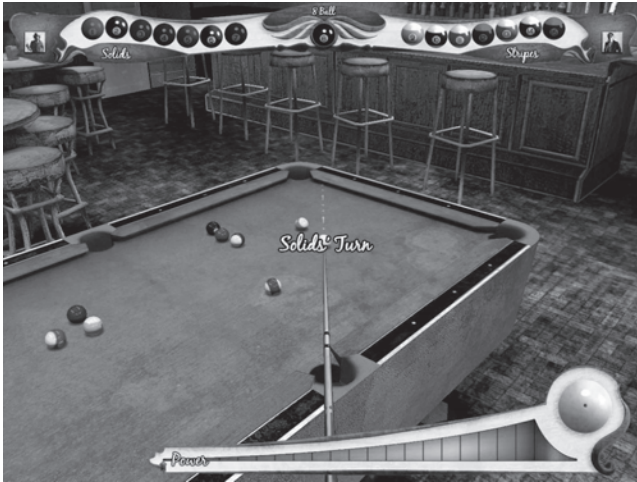


Figure 10-7: Screenshot from *Billiards Complete*

10.3 Installing OGRE 3D

Q. Fine; so OGRE is an acronym for Object-oriented Graphics Rendering Engine, and in summary it is a cross-platform and open-source SDK used primarily by game developers for making cross-platform, real-time 3D games. It sports a range of features from hierarchical scene management and particle systems to skeletal animation and image compositing. In addition, OGRE as an SDK has already been deployed by many developers to power a range of cross-platform 3D games, including *Ankh* and *Jettracer*. Having read about the basics of OGRE, then, from where can OGRE be downloaded and how is it installed to the system so it is ready to use in Code::Blocks for creating and compiling OGRE applications?

A. The download and installation process for OGRE differs between the Windows and Linux platforms. The OGRE installation procedures for these two operating systems are now considered below in more detail.



NOTE. An Internet connection is required to install OGRE and its related libraries and dependencies.

10.3.1 Downloading and Installing OGRE 3D on Ubuntu

1. Beginning from the Ubuntu desktop, launch the Synaptic Package Manager from the Ubuntu start menu by selecting **System | Administration | Synaptic Package Manager**.

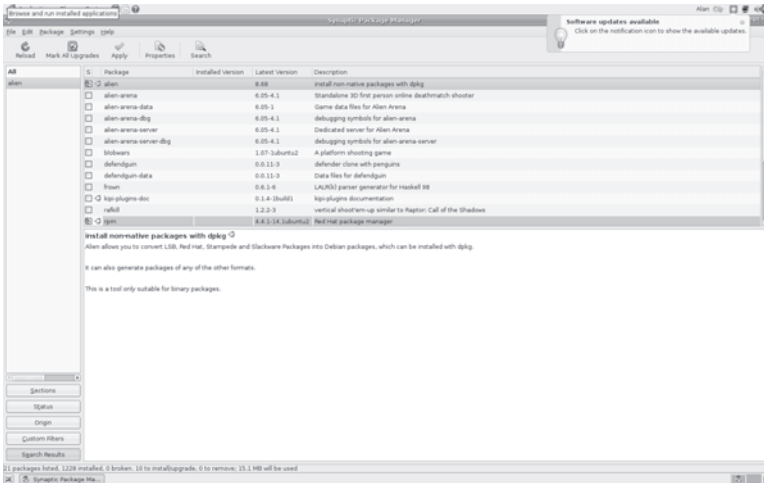


Figure 10-8:
Installing OGRE 3D

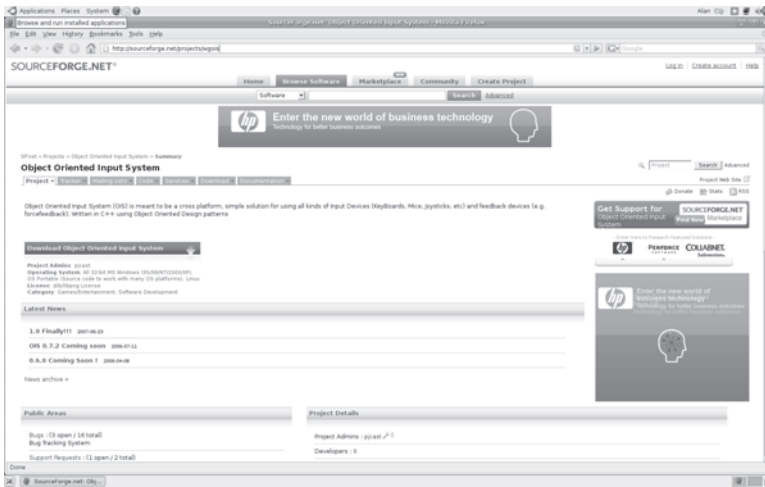
2. From the Synaptic Package Manager, search for and install the following packages:

alien	libxaw-header
autoconf	libxaw7-dev
automake1.9	ibfreetype6
automake1.6	libfreetype6-dev
build-essential	libpcre3
libc++punit-1.12-0	libpcre3-dev
libc++punit-dev	libzip-dev
libmng-dev	libxrandr-dev
libsdl1.2-dev	libxxf86vm-dev
libtool	freeglut3-dev



TIP. More installation details can be found in the OGRE 3D wiki and the OGRE online community at <http://www.ogre3d.org>.

3. After installing to the system each of the libraries listed in step 2, close the Synaptic Package Manager. Then from the Ubuntu desktop, navigate a web browser to download the OIS source code (Object-Oriented Input System) from <http://sourceforge.net/projects/wgois>. This library is used by OGRE for reading user input from peripheral devices, such as the mouse, keyboard, etc.



*Figure 10-9:
Downloading the
OIS source code*

4. Navigate a web browser to the FreeImage home page, and from there download the source distribution of their cross-platform image library, a lightweight library of functions for opening and managing common image file types, from JPEG and BMP to PNG and TGA. The source code for the FreeImage library can be found at <http://freeimage.sourceforge.net/>.

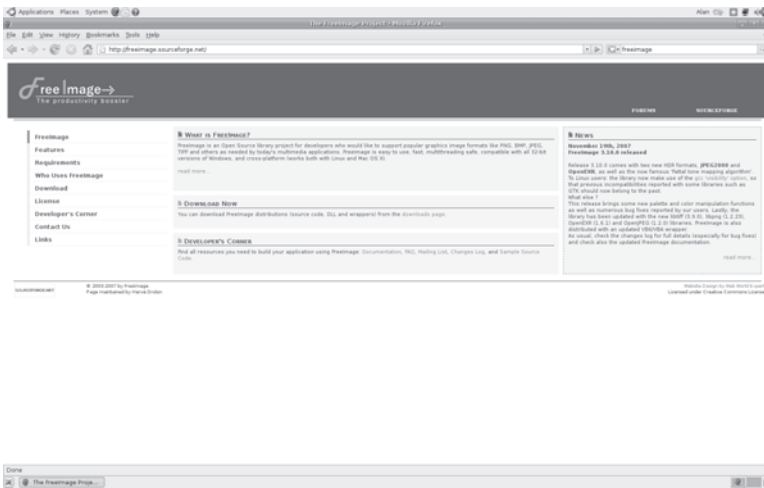


Figure 10-10: Downloading the FreeImage open-source library

- Navigate a web browser to http://www.cegui.org.uk/wiki/index.php/CEGUI_Downloads_0.5.0, and from there download the latest source distribution of Crazy Eddie's GUI, a cross-platform library used by OGRE to offer developers GUI facilities for their games.

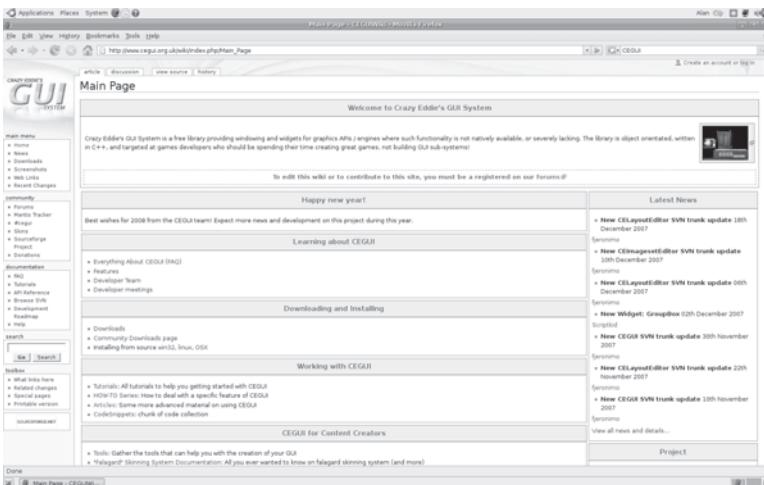


Figure 10-11: Downloading Crazy Eddie's GUI system

- Navigate a web browser to http://developer.nvidia.com/object/cg_toolkit.html, and from there download the latest CG pixel and vertex shader toolkit for cross-platform 3D games.

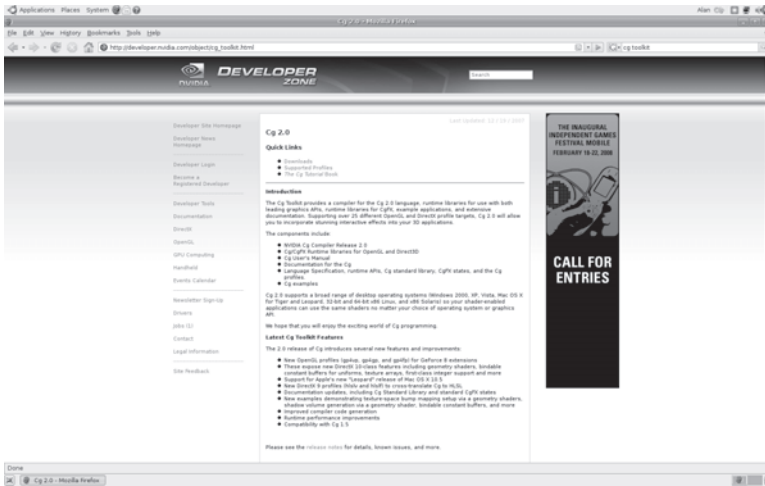


Figure 10-12:
Downloading the
latest NVIDIA toolkit

7. Then enter the following commands in the Ubuntu Terminal, pressing **Enter** after each line:

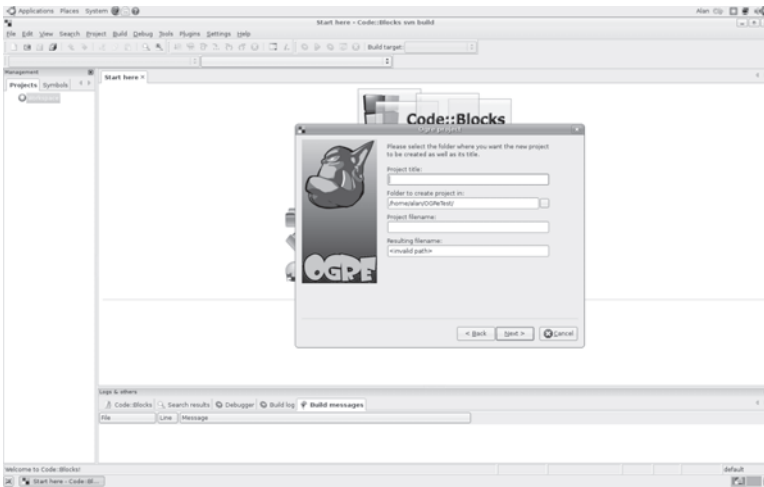

```
sudo alien Cg-1.5.i386.rpm
sudo dpkg -i cg_1.5.0-15_i386.deb
```
8. Navigate a web browser to the official OGRE 3D site at <http://www.ogre3d.org>, and from there download the latest OGRE source distribution for Linux.
9. Build the OIS, FreeImage, and Crazy Eddie's GUI packages downloaded in steps 3, 4, and 5 by extracting each package to a local directory, and from each directory run the following commands in a terminal window, pressing **Enter** after each line:

```
./bootstrap
./configure
./make
./install
```

10. Extract the contents of the OGRE 3D source archive (as downloaded from the OGRE 3D web site) to a local directory, and from there run the following command in an Ubuntu Terminal to compile and install the OGRE libraries to the system, ready for use in Code::Blocks:

```
aclocal ./bootstrap ./configure make sudo make checkinstall
```

11. Once OGRE 3D is installed to the system, launch the Code::Blocks C++ IDE from the Ubuntu main menu by selecting **Application | Programming | Code::Blocks**.
12. From the Code::Blocks Welcome display, begin a new project by selecting **File | New | Project** from the Code::Blocks main menu. Select an OGRE project. Click **Next** and follow the wizard.



*Figure 10-13:
Beginning a new
OGRE project from
the wizard*

13. A new OGRE Code::Blocks project is ready to compile and run.



NOTE. Some users may need to amend as appropriate the default library and header search paths used by the compiler when compiling and building new OGRE Code::Blocks projects. These settings are accessed from the Project Options menu, available by selecting **Project | Build Options** from the Code::Blocks main menu.

10.3.2 Downloading and Installing OGRE 3D on Windows

1. Beginning from the Windows desktop, navigate a web browser to the official OGRE 3D web page at <http://www.ogre3d.org>.



Figure 10-14

2. Click the **Download** button in the left margin of the page, choose **Download a prebuilt SDK**, then select the prebuilt binary SDK distribution **MinGW Code::Blocks** to download a copy of the OGRE SDK as a self-installer (EXE) from the web page to the local computer.



Figure 10-15: Downloading the MinGW prebuilt SDK

- Once completed, navigate a web browser to the Microsoft DirectX SDK home page at <http://www.microsoft.com/directx/> and download the latest DirectX 9 SDK distribution (if it is not installed already). Once downloaded, run the DirectX SDK installer.
- Run the OGRE 3D SDK Installer downloaded in step 2 of this section, and then follow the installation wizard to install OGRE 3D to the chosen directory on the local computer. When installation is complete, restart the computer.



Figure 10-16: Installing OGRE

- After restarting the computer, launch Code::Blocks (Nightly Build) from the Windows Start menu and open and compile the OGRE sample projects. Select **File | Open** and navigate to the **OGRE 3D SDK** folder, then open the **Samples** subfolder. Here, open the Samples Code::Blocks workspace, and use the Code::Blocks Build button from the toolbar to compile each sample project in the newly loaded sample workspace. These projects in both Debug and Release form are built and compiled ready-to-run in separate subfolders of the OGRE 3D SDK folder.

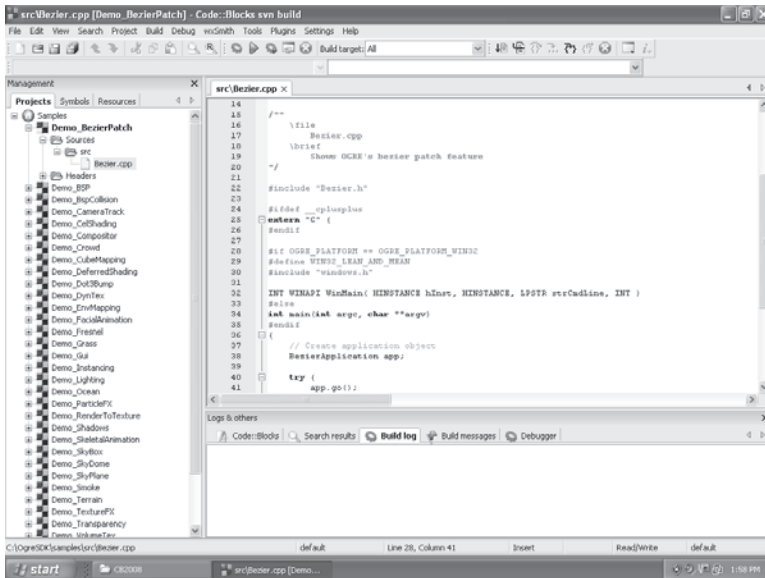


Figure 10-17:
Compiling each
sample project

- To create a new OGRE project, start Code::Blocks and select **File | New | Project** from the main menu of the Code::Blocks screen. When the New Project dialog appears, select **OGRE Project** from the project template list view. Click **OK** and then follow the wizard, after which a new OGRE project is ready to compile and run.

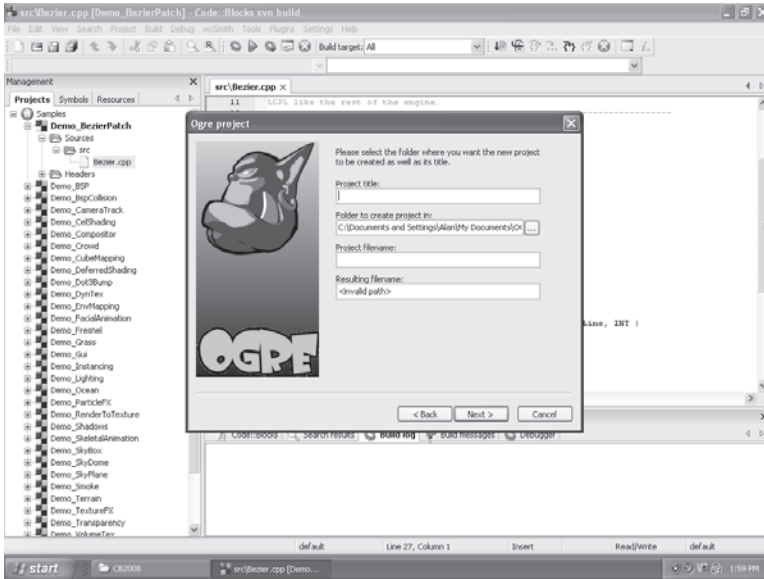


Figure 10-18:
Creating a new
OGRE project

10.4 Getting Started with OGRE 3D

Q. The OGRE SDK is now installed to the system, configured in Code::Blocks, and ready to compile to make OGRE applications. The Code::Blocks OGRE application wizard creates new OGRE projects, featuring all the necessary source code already generated in the source files, ready to build and run; this is the fundamental structure from which OGRE applications may be built. What more can you tell me about this OGRE framework? What does this generated code do? How exactly do OGRE applications work structurally?

A. Consider the following OGRE source code, the simplest OGRE application. The source code features annotations and highlights, which are discussed following the code.

```
//Links to libs: ogre_d, ogre, ois, ois_d
#ifdef OGRE_PLATFORM_WIN32
#define WIN32_LEAN_AND_MEAN
#include "windows.h"
#endif
```

```
#include "ExampleApplication.h"

#ifdef __cplusplus
extern "C" {
#endif

//-----
//Main application singleton object created at app startup
class MyFirstOGREApplication : public ExampleApplication
{
public:
    MyFirstOGREApplication() {}

protected:

    //Just override the mandatory create scene method
    //OnAppStart event
    //Do initialization here
    void createScene(void)
    {
        // {...}
    }

    void createFrameListener(void)
    {
        //Create an event listener object
        //That is, an object to receive an event on each frame

        // {...}
    }
};

//-----

#if OGRE_PLATFORM == OGRE_PLATFORM_WIN32
INT WINAPI WinMain(HINSTANCE hInst, HINSTANCE, LPSTR strCmdLine,
                  INT)
#else
int main(int argc, char **argv)
#endif
{
    // Create application object
    MyFirstOGREApplication app;
```

```
try {
    app.go(); //Start OGRE application
} 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;
}

#ifdef __cplusplus
}
#endif
```

As usual, application execution begins in the main (or WinMain) function, and here an instance of MyFirstOGREApplication is created, a user-defined OGRE class derived from class ExampleApplication. Representing the “game loop,” the “message pump,” or the lifetime of an OGRE application from beginning to end, the MyFirstOGREApplication class is instantiated in the main function. After instantiation, the method call `app.go()` initiates the game loop to step through each frame, and this method only returns as the application ends, terminated either by the user or by an error.

The user-defined `createScene` method of MyFirstOGREApplication is where game objects (scenes, enemies, classes, and files) are loaded, initialized, and configured, ready for use later in the game. This method is called automatically before the first game frame occurs, called by other methods working behind the scenes inherited from the base class ExampleApplication.

10.5 Receiving Frame Events

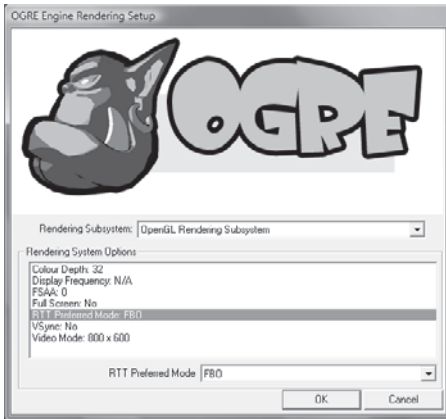


Figure 10-19: The OGRE setup screen is shown as an application begins

Q. The class `MyFirstOGREApplication` is therefore an application controller. Created at application startup, its `go` method signals the beginning of the self-sustaining game loop, and the return of this method marks the end of the loop whereupon the application may clean up and exit. The `MyFirstOGREApplication` class is self-sustaining and self-contained insofar as the game loop keeps itself alive “behind the scenes” (via methods defined in the ancestor class). The class calls its own methods (many defined in derived classes) at key events during the game loop, such as at scene startup and at application end. If the class is self-sustaining, then, how does it notify developers about frame events; for example, how can developers receive notifications on each frame to run typical game loop code such as reading user input, updating collision detection, or moving objects around a level?

A. Developers are notified about frame events via `FrameListener` classes; that is, classes derived from `FrameListener`, whose methods (such as `frameStarted` and `frameEnded`) are overridden and redefined in descendant classes, each of them called polymorphically once per frame. Consider the following code, amended from the previous sample to include a `FrameListener`:

```

#if OGRE_PLATFORM == OGRE_PLATFORM_WIN32
#define WIN32_LEAN_AND_MEAN
#include "windows.h"
#endif

#include "ExampleApplication.h"

#ifdef __cplusplus
extern "C" {
#endif

//-----

//OGRE frame listener class, called once per frame
class MyFirstListener : public ExampleFrameListener
{
protected:
public:
    MyFirstListener(RenderWindow* win, Camera* cam, const
        std::string &debugText)
        : ExampleFrameListener(win, cam)
        {
            mDebugText = debugText;
        }

    //Called once per frame
    bool frameStarted(const FrameEvent& evt)
    {
        if(ExampleFrameListener::frameStarted(evt) == false)
            return false;

        //Do stuff here {...}

        //Return 'false' to exit application, and 'true' to keep
        //alive game loop

        return true;
    }
};

//-----
//Main application singleton object created at app startup

```

```

class MyFirstOGREApplication : public ExampleApplication
{
public:
    std::string mDebugText;

    MyFirstOGREApplication() {}

protected:

    //Just override the mandatory create scene method
    //OnAppStart event
    //Do initialization here
    void createScene(void)
    {
        //{...}
    }

    void createFrameListener(void)
    {
        //Create an event listener object
        //That is, an object to receive an event on each frame

        mFrameListener= new MyFirstListener(mWindow, mCamera,
        mDebugText);

        //Add to list of listeners, each called once per frame
        //Can add more listeners if required
        //Though one listener is often more than enough
        mRoot->addFrameListener(mFrameListener);
    }
};

//-----

#if OGRE_PLATFORM == OGRE_PLATFORM_WIN32
INT WINAPI WinMain(HINSTANCE hInst, HINSTANCE, LPSTR strCmdLine,
    INT)
#else
int main(int argc, char **argv)
#endif
{
    // Create application object
    MyFirstOGREApplication app;

```

```
try {
    app.go(); //Start OGRE Application
} 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;
}

#ifdef __cplusplus
}
#endif
```

10.6 Adding Objects to a Scene

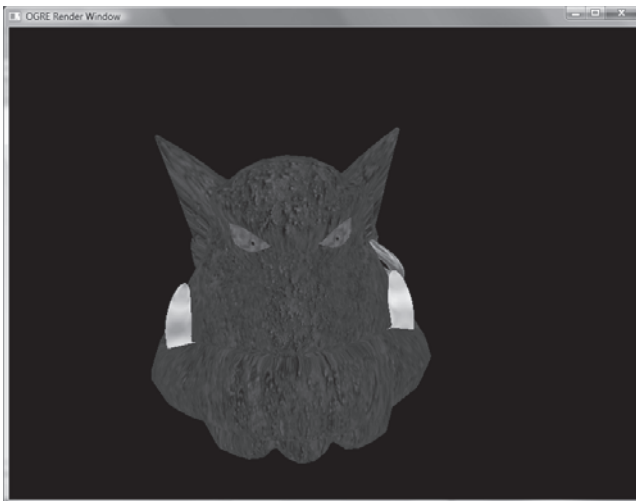


Figure 10-20: OGRE scene featuring one mesh

Q. Okay. So I create a new OGRE application using the Code::Blocks OGRE Wizard, and then I add an event listener class to receive an

event notification for each frame of the game loop at run time, frame by frame, until the user exits the application by pressing the Esc key on the keyboard. However, the newly created application features an empty scene; there are no objects, no lights, no materials, no animation — only blackness. How do I add objects to the game world (like a door, window, car, or NPC), and how do I assemble these to form a level in the game world?

A. In OGRE and 3D programming more generally, a 3D object (wall, door, NPC, etc.) is generically referred to as a “mesh,” and so an OGRE scene (or level, or world) is technically a geometric collection of meshes. It is a single Cartesian space containing an array of meshes similar to a 2D level that contains a collection of game objects arranged one in front of the other on a set of z-ordered layers. Chapter 7, “Game Mechanics,” further highlighted how objects in a scene are hierarchically connected to one another in terms of geometric properties. Each child object is affected by the position, orientation, and scale of its parent object in the game world. Move the parent object, and the child moves correspondingly, relative to its parent. Thus, a scene in OGRE refers to a Cartesian space containing a *hierarchy of geometrically related* meshes; in this space, each mesh is related to another in a parent-child-sibling relationship. In OGRE terminology: A *scene* is structurally a hierarchy of *nodes* (a spatial anchor, or the 3D equivalent of a 2D layer), where one or many meshes may be attached to a single node. All the meshes attached to a given node depend on the node for their position, rotation, and orientation. That is, the mesh is positioned relative to the node, and the position of the node affects the position of any attached meshes. The structural relationship between different meshes attached to different nodes reflects the relationship between the nodes themselves. Consider the following code taken from the `CreateScene` method of the OGRE application object. Here, the code creates a scene where a mesh is loaded from a file on disk, attached to a node in the scene, and then positioned in 3D space relative to the node at the origin (the *root node*).

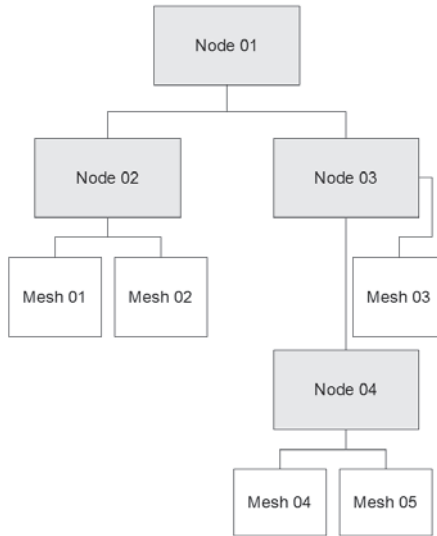


Figure 10-21: Node hierarchy

```
//[...] Other code here
```

```
class MyFirstOGREApplication : public ExampleApplication
```

```
{
```

```
public:
```

```
    std::string mDebugText;
```

```
    MyFirstOGREApplication() {}
```

```
protected:
```

```
    //Just override the mandatory create scene method
```

```
    //OnAppStart event
```

```
    //Do initialization here
```

```
void createScene(void)
```

```
{
```

```
    mSceneMgr->setShadowTechnique(SHADOWTYPE_TEXTURE_
        MODULATIVE);
```

```
    mSceneMgr->setShadowTextureSize(512);
```

```
    mSceneMgr->setShadowColour(ColourValue(0.6, 0.6, 0.6));
```

```
    // Set ambient light
```

```
    mSceneMgr->setAmbientLight(ColourValue(0.5, 0.5, 0.5));
```

```
Entity *ent = mSceneMgr->createEntity("head",
"ogrehead.mesh");

// Add entity to the root scene node
mSceneMgr->getRootSceneNode()->createChildSceneNode()->
attachObject(ent);
}

void createFrameListener(void)
{
//Create an event listener object
//That is, an object to receive an event on each frame

mFrameListener= new MyFirstListener(mWindow, mCamera,
mDebugText);

//Add to list of listeners, each called once per frame
//Can add more listeners if required
//Though one listener is often more than enough
mRoot->addFrameListener(mFrameListener);
}
};

//[...] Other code here
```

Inherited from `ExampleApplication`, `mSceneMgr` is an OGRE member accessible in the class `MyFirstOGREApplication`. `mSceneMgr` is the primary means of accessing and editing objects in an OGRE 3D scene, from creating scene nodes to positioning objects in 3D space.



NOTE. Meshes are loaded from files on disk and into OGRE as *entities*. As the mesh is loaded, OGRE searches its media paths for the specified file. The media paths for a given OGRE application are listed in the `resources.cfg` file, located in the same directory as the application executable.



TIP. Creating OGRE Meshes in Blender 3D 3D meshes are typically modeled in 3D rendering software (such as 3ds Max, Maya, or Blender 3D) and then exported from the modeling application to a file on disk. OGRE accepts meshes in the file format (.mesh). Although most applications cannot export to this format *natively*, there are many plug-ins available to extend the feature set of existing modeling software to export meshes to the OGRE format, including an OGRE plug-in for exporting Blender meshes.

The available OGRE exporters can be found at the OGRE wiki at http://www.ogre3d.org/wiki/index.php/OGRE_Exporters.

10.7 Adding Lights and Particle Systems

Q. Yes, I can now add meshes to an OGRE scene using nodes and entity hierarchies, but scenes often contain more than simply meshes. Scenes also feature lighting, shadows, particle systems (rain, snow, fog), and many other effects. How can I also add these to a scene?

A. Both lights and particle systems are added to scenes via scene nodes, in the same way meshes are attached to scene nodes. This is because meshes, lights, particle systems, and all other scene objects are derived from the same base class. Consider the following code to add both a particle system and lights to an OGRE scene:

```
//Main application singleton object created at app startup

class MyFirstOGREApplication : public ExampleApplication
{
public:
    std::string mDebugText;

    MyFirstOGREApplication() {}

protected:

    //Just override the mandatory create scene method
    //OnAppStart event
    //Do initialization here
```

```

void createScene(void)
{
    ColourValue mMinLightColour(0.5, 0.1, 0.0);
    ColourValue mMaxLightColour(1.0, 0.6, 0.0);

    mSceneMgr->setShadowTechnique(SHADOWTYPE_TEXTURE_
        MODULATIVE);
    mSceneMgr->setShadowTextureSize(512);
    mSceneMgr->setShadowColour(ColourValue(0.6, 0.6, 0.6));

    // Set ambient light
    mSceneMgr->setAmbientLight(ColourValue(0.5, 0.5, 0.5));

    Entity *ent = mSceneMgr->createEntity("head",
        "ogrehead.mesh");

    // Add entity to the root scene node
    mSceneMgr->getRootSceneNode()->createChildSceneNode()->
        attachObject(ent);

    mSceneMgr->getRootSceneNode()->createChildSceneNode()->
        attachObject(
    mSceneMgr->createParticleSystem("Fireworks",
        "Examples/Fireworks"));

    Light* mLight = mSceneMgr->createLight("Light2");
    mLight->setDiffuseColour(mMinLightColour);
    mLight->setSpecularColour(1, 1, 1);
    mLight->setAttenuation(8000,1,0.0005,0);

    // Create light node
    SceneNode *mLightNode = mSceneMgr->getRootSceneNode()->
        createChildSceneNode("MovingLightNode");
    mLightNode->attachObject(mLight);
}

void createFrameListener(void)
{
    //Create an event listener object
    //That is, an object to receive an event on each frame

    mFrameListener= new MyFirstListener(mWindow, mCamera,
        mDebugText);
}

```

```

//Add to list of listeners, each called once per frame
//Can add more listeners if required
//Though one listener is often more than enough
mRoot->addFrameListener(mFrameListener);
}
};

```



NOTE. Particle systems (such as Examples/Fireworks, as featured in the code sample above) are defined in OGRE scripts, and these can be found in the OGRE Samples/Scripts subdirectory. Here is a sample particle system script:

```

// Exudes greeny particles that float upwards
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 them float upwards
    affector LinearForce

```

```
{
    force_vector      0 100 0
    force_application add
}

// Fader
affector ColourFader
{
    red -0.25
    green -0.25
    blue -0.25
}
}
```

10.8 Reading User Input with OGRE and OIS

Q. I have so far seen how to load objects like meshes from files on disk and into an OGRE scene as entities, how to position those objects relative to scene nodes, and also how to create lighting and particle systems for special effects like rain, snow, and volumetric fog. But none of this is much good for making computer games generally unless I can piece together instructions from the user by reading their input from peripheral devices (like a keyboard or mouse). So, how can I determine which key is pressed on the keyboard; for example, whether or not the user is holding the down arrow or up arrow? Or how can I determine the X,Y coordinate of the mouse cursor on-screen?

A. To read user input (both keyboard and mouse) through OGRE, the OGRE application class should be derived from multiple classes (not just `ExampleApplication`) to support a new set of inherited methods (event handlers) that are called whenever input events occur. The following code features a full sample OGRE application, and it both summarizes the code covered throughout this chapter and highlights in detail how OGRE handles user input.



NOTE. OGRE input key codes can be found in Appendix J.

```

#if OGRE_PLATFORM == OGRE_PLATFORM_WIN32
#define WIN32_LEAN_AND_MEAN
#include "windows.h"
#endif

#include "ExampleApplication.h"

#ifdef __cplusplus
extern "C" {
#endif

//-----

//OGRE frame listener class, called once per frame
class MyFirstListener : public ExampleFrameListener, public
OIS::KeyListener, public OIS::MouseListener
{
protected:
public:

//-----//

    MyFirstListener(RenderWindow* win, Camera* cam, const
        std::string &debugText)
        : ExampleFrameListener(win, cam)
        {
            mDebugText = debugText;

mMouse->setEventCallback(this);
mKeyboard->setEventCallback(this);
        }

//-----//

    //Called once per frame
    bool frameStarted(const FrameEvent& evt)
    {

        if(ExampleFrameListener::frameStarted(evt) == false)
            return false;

        //Exits application when down arrow key is pressed

```



```
        if(mKeyboard->isKeyDown(OIS::KC_DOWN))
            return false;

        //Do stuff here {...}

        //Return "false" to exit application, and "true" to keep
        // alive game loop

        return true;
    }

    bool mouseMoved(const OIS::MouseEvent &arg)
    {
        return true;
    }

    //-----//
    bool mousePressed(const OIS::MouseEvent &arg, OIS::
MouseButtonID id)
    {
        return true;
    }

    //-----//
    bool mouseReleased(const OIS::MouseEvent &arg, OIS::
MouseButtonID id)
    {
        //If left button pressed
        //0 = left
        //1 = right
        //2 = middle
        if(id == 0)
        {
            //Do stuff here
        }

        //If mouse X pos has not moved
        if(arg.state.X.rel==0)
        {
            //Do stuff here
        }
    }
}
```

```
        return true;
    }

//-----//
    bool keyPressed(const OIS::KeyEvent &arg)
    {
        //If Escape key pressed
        if(arg.key == OIS::KC_ESCAPE)
        {
            //Do stuff here
        }

        return true;
    }

//-----//
    bool keyReleased(const OIS::KeyEvent &arg)
    {
        return true;
    }
};

//-----//
//Main application singleton object created at app startup

class MyFirstOGREApplication : public ExampleApplication
{
public:
    std::string mDebugText;

//-----//

    MyFirstOGREApplication() {}

protected:

    //Just override the mandatory create scene method
    //OnAppStart event
    //Do initialization here

//-----//
```

```

void createScene(void)
{
    //{...}
}

//-----//

void createFrameListener(void)
{
    //Create an event listener object
    //That is, an object to receive an event on each frame

    mFrameListener= new MyFirstListener(mWindow, mCamera,
        mDebugText);

    //Add to list of listeners, each called once per frame
    //Can add more listeners if required
    //Though one listener is often more than enough
    mRoot->addFrameListener(mFrameListener);
}
};

//-----//

#if OGRE_PLATFORM == OGRE_PLATFORM_WIN32
INT WINAPI WinMain(HINSTANCE hInst, HINSTANCE, LPSTR
    strCmdLine, INT)
#else
    int main(int argc, char **argv)
#endif
{
    // Create application object
    MyFirstOGREApplication app;

    try {
        app.go(); //Start OGRE Application
    } 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);
        #endif
    }
}

```

```
        #else
            std::cerr << "An exception has occurred: " <<
                e.getFullDescription();
        #endif
    }

    return 0;
}

#ifdef __cplusplus
}
#endif
```

10.9 Conclusion

OGRE (Object-oriented Graphics Rendering Engine) is perhaps one of the single largest and most comprehensive open-source 3D SDKs available. As such, this chapter can be considered little more than an OGRE introduction; there is far more to OGRE than what is included here. Consequently, perhaps the best strategy is to now sort through the OGRE sample applications shipped with the OGRE SDK, examining each application and its code and scripts and making changes to observe their effects. The ever-growing OGRE wiki is also a comprehensive source of OGRE information, written by OGRE users for OGRE users.

This page intentionally left blank.

Chapter 11

Other Cross-Platform SDKs and Tools

From C++ games compiled in Code::Blocks to Shockwave games created in Adobe Director, and from 2D games via SDL and Novashell to 3D games via OGRE, this book has generally considered three primary kinds of *cross-platform* games. First, we considered those cross-platform games that are first coded by developers in C++ through the Code::Blocks IDE, and then later cross-compiled to run *natively* on each target platform, whether it be Linux, Mac, or Windows. Second, we considered engine-based cross-platform games where a cross-platform GUI editor (like Novashell) is used to create a platform-independent game image (featuring the levels, NPCs, and scripting); overall a game image finally interpreted and executed by a natively compiled cross-platform engine so the game runs “as good as” natively. And similar to the second kind, the third kind includes the VM image partnership of games (like Shockwave games) typically compiled into a game image through a GUI editor (like Director) and then run through a virtual machine to run as a web game in a browser or as a stand-alone executable. This final chapter of the book now considers briefly a whole series of alternative cross-platform SDKs and tools available for game development, each of them worthy of further consideration by developers who have specific developmental requirements or who wish to use SDKs other than those featured earlier in this book. Specifically, we look at a series of SDKs that may be categorized as one of the following kinds:

- **Graphics SDK** — Graphics SDKs like SDL (Simple DirectMedia Layer) and OGRE (Object-oriented Graphics Rendering Engine) are largely concerned with presenting real-time graphics to the

game window, whether 2D or 3D graphics. As this book has hopefully shown, such APIs offer tools and classes designed specifically for loading images from files on disk and into system (or other hardware) memory as resources, ready for display or animation in the game window.

- **Audio SDK** — Audio SDKs (such as `SDL_mixer`, `FMOD`, and `BASS`) are libraries featuring classes and tools used by developers to play audio (music and sound effects) via the audio hardware to the speakers.
 - **Physics SDK** — GUI editors such as `Novashell` and `Adobe Director` offer to game developers a whole subset of game editing and level designing features to create comprehensive game worlds in which exist buildings, NPCs, and all kinds of other objects and phenomena, many of which can be found in the “real” world. The purpose of a physics SDK is to offer to developers a mathematical framework of functions and classes designed to simulate “real life” physics so game objects and game worlds may behave like those in the real world. It does all the computational hard work for you automatically.
 - **Network SDK** — Games described as multiplayer (such as `Unreal Tournament`, `Quake`, and `World of Warcraft`) are those that bring together into a single online social space thousands of gamers from many disparate regions around the globe, each of them meeting up with others to play their games both competitively (e.g., death-match) or cooperatively (e.g., team death-match). Network SDKs include some of the libraries and tools that make it possible for game developers to create games that talk to one another across the Internet. These games establish mutual socket connections, and through these transmit data to and from each other to synchronize multiplayer facilities.
 - **Artificial Intelligence SDK** — Artificial intelligence (AI) refers to the processes (the set of functions and algorithms) that make computers think for themselves, or appear to think for themselves. Through AI, computers can play chess, control NPCs, navigate NPCs intelligently through levels by avoiding obstacles and traveling the shortest route between any two points in a map, and engage players in combat in real-time and turn-based strategy
-

games. A cross-platform AI SDK, then, offers to developers the thinking, calculating, and cognitive apparatus (functions, classes, and toolsets) to implement AI into their cross-platform games in order to make their games think, reason, and respond.

- **Input SDK** — Cross-platform input SDKs (like OIS used by OGRE) boast a set of platform-independent functions and classes that allow developers to read user input from input peripherals such as keyboards, mice, and joysticks.
- **Scripting SDK** — Game engines like Novashell, graphics engines like OGRE, and Shockwave games made in Director make use of scripting facilities; that is, developers use a scripting language such as Lua, Python, or JavaScript to code and edit a game’s behavior without needing to recompile the entire source code from scratch. Thus, scripting SDKs offer the bridging tools (the functions and classes) to bridge the technical gap between the binary executable and a script in a file.
- **Game Engine** — Game engines like Novashell (an interpreter of platform-independent game images) are typically designed as a “one-stop” complete game development solution. That is, game engines typically market themselves as being a synthesis of libraries, a “feature complete” totality insofar as they offer graphics rendering, audio playback, physics and AI, scripting facilities for customization, file input/output, level editors, and sometimes a network multiplayer feature set.
- **GUI SDK** — The term GUI (graphics user interface) refers to the widgets, gadgets, and gizmos (like buttons, list views, edit boxes, and other interface components) found on game screens. A GUI SDK, then, offers a set of easy-to-use tools and classes for creating in-game GUIs for cross-platform games.
- **Web SDK** — Web SDK is a broad term used to designate a whole range of SDKs that promote gaming online. This chapter will consider YABB, an online forum kit game developers can use to create an online community for their gamers, where they can hang out, socialize with other like-minded and not-so-like minded gamers, discuss hints and tips for specific sections of games, fight and argue, report game bugs and potential issues, and finally offer praise or words of encouragement to the developers.

- **Distribution SDK** — Generally, contemporary game developers distribute their games to users either as a published CD/DVD sold commercially or as a self-published online download (either directly from the developer’s web site or through an online gaming portal like Reflexive Arcade or Big Fish Games). But whether the game comes in a boxed CD/DVD or as an online download, the game typically installs itself to the user’s computer via an automated installer. This chapter examines some means of creating game installers using a distribution SDK.

11.1 Graphics SDKs

11.1.1 OpenGL

OpenGL (Open Graphics Library) is a cross-language, cross-platform SDK for fast-paced (hardware-accelerated) 2D and 3D computer graphics. The library has over 250 different function calls to draw complex three-dimensional objects and scenes, ranging from simple primitives to complex animated geometry. OpenGL was developed by Silicon Graphics, Inc. (SGI) in 1992 and is widely used in CAD, virtual reality, scientific visualization, information visualization, and flight simulation, as well as in video games.

Platforms supported: Windows, Linux, and Mac

Web site: <http://www.opengl.org/>

License: Free for commercial and non-commercial use.

11.1.2 PTK

The PTK web site describes PTK as “a multi-platform 2D game engine with 3D capabilities built around OpenGL or Direct 3D accelerated hardware, however, it is also possible to create 3D multi-platform games with OpenGL [...] PTK can be used by a wide variety of users: from the most experienced programmers to the newbie aspiring game programmers.”

Platforms supported: Mac and Windows

Web site: <http://www.phelios.com/ptk/>

License: Free only for non-commercial use; more license details available at their web site.

11.1.3 ClanLib

Freely available, high-level, and cross-platform, ClanLib is an OpenGL-powered open-source SDK for creating cross-platform 2D games using the C++ language. ClanLib boasts a variety of features, some of which include XML/DOM support, 2D collision detection, network library, sound mixer supporting WAV, tracker formats (mod/s3m/xm/...) and ogg-vorbis, and high-level 2D graphics API supporting OpenGL and SDL as render targets.

Platforms supported: Windows, Mac, and Linux

Web site: <http://www.clanlib.org/>

License: BSD-style; free for commercial and non-commercial use. See web site for more information.

11.1.3.1 Installing ClanLib

It has on occasion been said that ClanLib is a troublesome SDK to install and configure in terms of downloading the SDK from the web, unpacking the downloaded SDK package, and getting set up in order to code and successfully compile ClanLib applications in a C++ IDE like Code::Blocks, and is especially troublesome on Linux. The following step-by-step installation guide details how to install and compile ClanLib applications on Linux Ubuntu.

1. Beginning from the Ubuntu desktop, open a Terminal window by choosing **Applications | Accessories | Terminal**.
2. Download the required ClanLib libraries by entering the following terminal commands:

```
sudo apt-get install zlib1g-dev libjpeg62-dev libpng12-dev  
libmikmod2-dev libogg-dev libvorbis-dev libxxf86vm-dev
```

3. Close the Terminal window and navigate a web browser to the ClanLib web site at <http://www.clanlib.org/>. Here, download to the local computer the ClanLib source code package for Linux, `ClanLib-0.8.0.tgz`. Unpack the contents of this package to a directory on the local machine.
4. In this local directory, open a Terminal window and run the following commands by pressing the **Enter** key after typing each line:

```
tar xvzf ClanLib-0.8.0.tgz
cd ClanLib-0.8.0
./configure
make
sudo make install
```

5. Open with administrator privileges (write permission) the local system file `/etc/ld.so.conf`, and add to the end of this file the following line, then choose **File | Save**.

```
include /usr/local/lib/
```

6. Return to the Terminal and enter the following commands, pressing **Enter** after each line:

```
sudo ldconfig
export PKG_CONFIG_PATH=/usr/local/lib/pkgconfig
```

7. ClanLib is now installed to the local machine, ready to work with Code::Blocks in compiling and running newly created ClanLib applications. Start Code::Blocks and create a blank new project. Into the main source file paste the following ClanLib code:

```
#include <ClanLib/gl.h>
#include <ClanLib/core.h>
#include <ClanLib/application.h>
#include <ClanLib/display.h>

class MyApp : public CL_ClanApplication
{
public:
    virtual int main(int argc, char **argv)
    {
        // Create a console window for text output if not available
        // Use printf or cout to display some text in your program
        CL_ConsoleWindow console("Console");
```

```
console.redirect_stdio();

try
{
    // Initialize ClanLib base components
    CL_SetupCore setup_core;

    // Initialize the ClanLib display component
    CL_SetupDisplay setup_display;

    // Initialize the ClanLib GL component
    CL_SetupGL setup_gl;

    // Create a display window
    CL_DisplayWindow window("ClanLib application", 640, 480);

    // Run until someone presses Escape
    while (!CL_Keyboard::get_keycode(CL_KEY_ESCAPE))
    {
        // Clear the display in a dark blue shade
        // The four arguments are red, green, blue and alpha
        // (defaults to 255)
        // All color shades in ClanLib are measured in the
        // range 0-255
        CL_Display::clear(CL_Color(0, 0, 50));

        // Flip the display (using a double buffer),
        // showing on the screen what we have drawn
        // since last call to flip()
        CL_Display::flip();

        // This call updates input and performs other
        // "housekeeping"
        // Call this each frame
        // Also, gives the CPU a rest for 10 milliseconds
        // to catch up
        CL_System::keep_alive(10);
    }
}
// Catch any errors from ClanLib
catch (CL_Error err)
{
    // Display the error message
```

```
        std::cout << err.message.c_str() << std::endl;
    }

    // Display console close message and wait for a key
    console.display_close_message();

    return 0;
}
} app;
```

8. Compile and run the ClanLib code.

11.2 Audio SDKs

11.2.1 FMOD

FMOD is a cross-platform commercial audio library made by Firelight Technologies that plays audio files in the following formats: AIFF, ASF, ASX, DLS, FLAC, FSB, IT, M3U, MID, MOD, MP2, MP3, OGG, PLS, RAW, S3M, VAG, WAV, WAX, WMA, XM, and XMA.

Platforms supported: Windows, Mac, Linux, Nintendo GameCube, Wii, Solaris, Xbox, Xbox 360, PlayStation 2, PlayStation Portable, and PlayStation 3

Web site: <http://www.fmod.org/>

License: Free only for non-commercial use; more license details available at their web site.

11.2.2 BASS

BASS is a commercial cross-platform audio SDK by Un4seen Developments supporting audio files in the following file formats: WAV, AIFF, MP3, MP2, MP1, OGG, XM, IT, S3M, MOD, MTM, and UMX.

Platforms supported: Windows and Mac

Web site: <http://www.un4seen.com/>

License: Free only for non-commercial use; more license details available at their web site.

11.2.3 irrKlang

irrKlang is a commercial cross-platform audio SDK by Ambiera supporting audio files in the following file formats: WAV, MP3, OGG, XM, IT, S3D, and MOD.

Platforms supported: Windows, Linux, and Mac

Web site: <http://ambiera.com/irrklang/>

License: Free only for non-commercial use; more license details available at their web site.

11.2.4 Audiere

A free, open-source, and cross-platform audio SDK, Audiere supports audio files in the following file formats: WAV, AIFF, MP3, MP2, MP1, OGG, XM, IT, S3M, MOD, MTM, and UMX.

Platforms supported: Windows, Mac, and Linux

Web site: <http://audiere.sourceforge.net/>

License: LGPL; free for commercial and non-commercial use.

11.2.5 OpenAL

OpenAL is a generally free, open-source, and cross-platform audio SDK.

Platforms supported: Windows, Mac, Linux, BSD, Solaris, IRIX, Xbox, and Xbox 360

Web site: <http://www.openal.org/>

License: LGPL; free for commercial and non-commercial use.

11.3 Physics SDKs

11.3.1 ODE

ODE (Open Dynamics Engine) is a free, open-source, and cross-platform physics SDK for simulating both rigid body physics and collision detection. It has powered many games, including BloodRayne 2, Call of Juarez, and S.T.A.L.K.E.R.

Platforms supported: Linux, Windows, and Mac

Web site: <http://www.ode.org/>

License: BSD-style; free for commercial and non-commercial use.

11.3.2 Newton Game Dynamics

Newton Game Dynamics is a cross-platform physics SDK, and, according to the web site, is “an integrated solution for real-time simulation of physics environments. The API provides scene management, collision detection, [and] dynamic behavior and yet it is small, fast, stable, and easy to use.”

Platforms supported: Windows, Mac, and Linux

Web site: <http://www.newtondynamics.com/>

License: Free with restrictions. See web site for further details.

11.3.3 True Axis Physics

True Axis is a cross-platform physics SDK featuring collision detection, scene management, joints and rigid body dynamics, and contact force computation.

Platforms supported: Windows and Linux

Web site: <http://www.trueaxis.com/>

License: Free only for non-commercial use; more license details available at their web site.

11.3.4 **OPAL**

OPAL (Open Physics Abstraction Layer) is a free, open-source, and cross-platform physics SDK featuring linear and angular motion damping, collision detection, sensors, joints, and more.

Platforms supported: Windows, Mac, and Linux

Web site: <http://opal.sourceforge.net/>

License: LGPL; free for commercial and non-commercial use.

11.3.5 **Bullet**

Bullet is a free, open-source, and cross-platform physics SDK featuring linear and angular motion damping, collision detection, sensors, joints, and more.

Platforms supported: Mac, Windows, Linux, and PlayStation 3

Web site: <http://www.continuousphysics.com/Bullet/>

License: ZLib; free for commercial and non-commercial use.

11.3.6 **PhysX**

PhysX is a commercial, cross-platform physics SDK used by many games including Unreal.

Platforms supported: Mac, Windows, Linux, and consoles

Web site: <http://www.ageia.com/>

License: Free only for non-commercial use; more license details available at their web site.

11.4 Network SDKs

11.4.1 RakNet

According to the RakNet web site, “RakNet is a networking API that is a wrapper for reliable UDP and higher level functionality on Windows, Linux, and Unix. It allows any application to communicate with other applications on the same computer, over a LAN, or over the Internet. Although it could be used for any networked application, it was developed specifically for rapid development of online games and the addition of multiplayer to single-player games.”

Platforms supported: Windows and Linux

Web site: <http://freshmeat.net/projects/raknet>

License: Free with restrictions; more license details available at their web site.

11.4.2 HawkNL

HawkNL is a free, cross-platform, open-source, “game-oriented” network API designed largely as a wrapper over Berkeley/Unix Sockets and Winsock. HawkNL also provides other features including support for many groups of sockets, socket statistics, high-accuracy timer, CRC functions, macros to read and write data to packets with endian conversion, and support for multiple network transports.

Platforms supported: Windows, Linux, Mac, IRIX, AIX, BSD, and Solaris

Web site: <http://www.hawksoft.com/hawknl/>

License: LGPL; free for commercial and non-commercial use.

11.4.3 **SDL_net**

SDL_net is a free, open-source, cross-platform SDL networking extension library for the SDL library.

Platforms supported: Linux, Windows, BeOS, Mac OS, Mac OS X, FreeBSD, NetBSD, OpenBSD, BSD/OS, Solaris, IRIX, and QNX

Web site: http://www.libsdl.org/projects/SDL_net/

License: LGPL; free for commercial and non-commercial use.

11.5 **Artificial Intelligence SDKs**

11.5.1 **Boost Graph Library**

Boost is a series of open-source, cross-platform, and peer reviewed C++ libraries, including the graph library used by many for pathfinding and other AI game development purposes.

Platforms supported: Windows, Linux, Mac, and consoles

Web site: http://boost.org/libs/graph/doc/table_of_contents.html

License: Free; more license details available at their web site.

11.5.2 **OpenSteer**

According to the web site, “OpenSteer is a C++ library to help construct steering behaviors for autonomous characters in games and animation. In addition to the library, OpenSteer provides an OpenGL-based application called OpenSteerDemo, which displays predefined demonstrations of steering behaviors. The user can quickly prototype, visualize, annotate, and debug new steering behaviors by writing a plug-in for OpenSteerDemo.”

Platforms supported: Linux, Windows, BeOS, Mac OS, Mac OS X, FreeBSD, NetBSD, OpenBSD, BSD/OS, Solaris, IRIX, and QNX

Web site: <http://opensteer.sourceforge.net/>

License: MIT; free for commercial and non-commercial use.

11.5.3 FANN

FANN (Fast Artificial Neural Network) is an open-source, cross-platform AI library that “implements multilayer artificial neural networks in C with support for both fully connected and sparsely connected networks.”

Platforms supported: Windows, Mac, and Linux

Web site: <http://leenissen.dk/fann/>

License: LGPL; free for commercial and non-commercial use.

11.5.4 Garfixia AI Repository

This is a free, cross-platform, and open-source collection of common AI functions, classes, and algorithms.

Platforms supported: Linux, Windows, BeOS, Mac OS, Mac OS X, FreeBSD, NetBSD, OpenBSD, BSD/OS, Solaris, IRIX, and QNX

Web site: <http://www.dossier-andreas.net/ai/index.html>

License: Free only for non-commercial use; more license details available at their web site.

11.6 Input SDKs

11.6.1 LibGII

A free, cross-platform, and open-source input management library, LibGII features functions and classes to read user input from peripheral input devices, including mouse, keyboard, joysticks, and others.

Platforms supported: Linux, Windows, Mac OS, Mac OS X, FreeBSD, and OpenBSD

Web site: <http://www.ggi-project.org/packages/libgii.html>

License: MIT; free for commercial and non-commercial use.

11.6.2 OpenInput

OpenInput, according to the web site, “is a free, open-source, cross-platform, and portable input handling library written in C. The library can take input from several devices like mice, joysticks, and keyboards, and presents it to the user using a simple, platform-independent, and easy-to-use API.”

Platforms supported: Windows and Linux

Web site: <http://home.gna.org/openinput/>

License: LGPL; free for commercial and non-commercial use.

11.7 Scripting SDKs

11.7.1 Lua

Created in 1993 by Roberto Ierusalimschy, Lua (pronounced Loo-ah) is a free, cross-platform, and open-source imperative procedural scripting language used by many games including World of Warcraft, SimCity 4, Crysis, and Supreme Commander. It is also used by Novashell as well as other game engines and game editors.

Platforms supported: Windows, Linux, Mac, BREW, Symbian, and PocketPC

Web site: <http://www.lua.org/>

License: MIT; free for commercial and non-commercial use.

11.7.2 Python

According to the Python web site, “Python is a dynamic, object-oriented programming language that can be used for many kinds of software development. It offers strong support for integration with other languages and tools, comes with extensive standard libraries, and can be learned in a few days. Many Python programmers report substantial productivity gains and feel the language encourages the development of higher quality, more maintainable code.”

Platforms supported: Windows, Mac, Linux, Amiga, Palm handhelds, and Nokia mobile phones

Web site: <http://www.python.org/>

License: Free for commercial and non-commercial use.

11.7.3 Ruby

Ruby is a free, open-source, cross-platform, object-oriented scripting language.

Platforms supported: Windows, Linux, and Mac

Web site: <http://www.ruby-lang.org/>

License: Free for commercial and non-commercial use.

11.7.4 Squirrel

Squirrel is a free, open-source, cross-platform, object-oriented scripting language, some of whose features include dynamic typing, exception handling, classes and inheritance, tail recursion, and automatic memory management.

Platforms supported: Windows, Mac, and Linux

Web site: <http://squirrel-lang.org/>

License: Free for commercial and non-commercial use.

11.7.5 AngelCode

AngelCode is a free, open-source, cross-platform, object-oriented scripting language.

Platforms supported: Windows, Linux, Mac OS X, Xbox, Xbox 360, PlayStation 2, PlayStation Portable, PlayStation 3, Dreamcast, Nintendo DS, and Windows mobile

Web site: <http://www.angelcode.com/angelscript/>

License: Free for commercial and non-commercial use.

11.7.6 GameMonkey

GameMonkey, according to the web site, “is an embedded scripting language that is intended for use in game and tool applications. GameMonkey is, however, suitable for use in any project requiring simple scripting support. GameMonkey borrows concepts from Lua (www.lua.org), but uses syntax similar to C, making it more accessible to game programmers. GameMonkey also natively supports multithreading and the concept of states.”

Platforms supported: Windows, Mac, and Linux

Web site: <http://www.somedude.net/gamemonkey/>

License: MIT license; free for commercial and non-commercial use.

11.8 Game Engines

11.8.1 Torque

Torque is a commercial game engine, complete with level editor, sound, input, graphics renderer, and more.

Platforms supported: Windows, Mac, and Linux

Web site: <http://www.garagegames.com/>

License: Commercial

11.8.2 Irrlicht

Irrlicht Engine is a free, open-source, real-time 3D engine written in C++. Cross-platform (using D3D, OpenGL, and its own software renderer), Irrlicht has a huge active community where you can find enhancements such as terrain renderers, portal renderers, exporters, world layers, tutorials, editors, and language bindings for Java, Perl, Ruby, Basic, Python, Lua, and so on.

Platforms supported: Windows, Mac, and Linux

Web site: <http://irrlicht.sourceforge.net/>

License: Zlib; free for commercial and non-commercial use.

11.8.3 Game Editor

Game Editor is aimed at those new to game programming. Using Game Editor, developers can build 2D games for the PC and mobile phone platforms.

Platforms supported: Windows, Mac, and Linux

Web site: <http://game-editor.com/>

License: Commercial

11.9 GUI SDKs

11.9.1 OpenGUI

OpenGUI is an open-source, cross-platform, and freely available C++ GUI framework for games and other cross-platform applications.

Platforms supported: Windows, Mac, and Linux

Web site: <http://opengui.rightbracket.com/index.php>

License: BSD; free for commercial and non-commercial use.

11.10 Web SDKs

11.10.1 YaBB

YaBB is a free, open-source forum software package also offering a real-time chat and support system for your web site visitors. Using YaBB, game developers can build a technical support and social community for their gamers.

Platforms supported: Windows, Mac, and Linux

Web site: <http://www.yabbforum.com/>

License: Free for commercial and non-commercial use.

11.10.1.1 Downloading, Installing, and Creating an Online Forum

The following step-by-step guide illustrates how to download, install, and create an online forum, which can be used to build gaming communities. Forums provide a place where gamers can log on and speak to other gamers, report bugs, provide help, and communicate with developers.

1. Beginning from the desktop (Win, Mac, or Linux), navigate a web browser to the YaBB home page at <http://www.yabbforum.com/>.
2. At the YaBB home page, click the **Downloads** menu item to display the downloads page. From there, select one of the two YaBB packages listed: YaBB_2.2.zip or YaBB_2.2.tar.gz. Download the archive from the web to the local machine, and then extract the archive to a local directory.
3. Then download the free, cross-platform, and open-source FTP client FileZilla by navigating a web browser to the FileZilla home page at <http://filezilla-project.org/>.
4. At the FileZilla web site, download and install the FileZilla FTP client application from the web site to the local computer.
5. Return to the directory containing the contents of the extracted YaBB package. This directory includes the following files and directories:
 - **cgi-bin** — This directory contains the core script files for the forum. This eventually will be uploaded to the web server.
 - **public_html** — This directory features HTML pages. These are the default pages for the forum. They do not need to be edited, but it is possible to do so if you wish to change the look of your forum.
 - **Quick-Guide** — The help documentation.

6. Using the FileZilla FTP software, log onto the web host using the FTP server (e.g., ftp.mywebsite.com). Provide a user name and password as appropriate. Please remember that YaBB can only be used on web hosts that support Perl scripts. Most commercial hosts do, but many free hosts do not.
7. On the web space, find a directory called cgi-bin. If it does not exist, then a directory with this name should be created. From the local computer, copy all the contents of cgi-bin (inside the directory where YaBB was extracted) to the cgi-bin directory on the web host.
8. Outside of the cgi-bin directory on the web host, a new folder should be created, usually in public_html. This folder should reside among the rest of the standard HTML pages and files for the web site. Ideally, this should be called “yabbfiles,” but the name can be anything desired. From the local computer, copy the yabbfiles directory (inside public_html where YaBB was extracted). This should be copied to the web host inside the newly created yabbfiles directory.
9. Once the required files have been copied to the web host, the access privileges for these files and directories must be changed. These settings affect whether the files can be written to and read from.

At this point, all the forum files are uploaded to the server. Each file on the server has a series of permissions for reading and writing, and also for whether the file can be accessed at all. YaBB requires the forum files to be changed to specific settings in order to work. Each setting, such as read or write, has an integer cost, and the total cost for any file reflects the range of settings that apply to the file. This is sometimes called chmod. The quickest method for setting permissions is to use FileZilla FTP, which allows you to select the file, enter a specific integer, and then click Apply.

- a. Right-click one or more files and/or directories in the browser.
 - b. Click **File Attributes**.
-

- c. This displays a menu where a value can be entered. This value reflects the combination of access privileges to be applied to the file(s).

The following table lists the YaBB files and directories to be changed and their corresponding privilege values.

CGI-BIN files:

```

chmod 755: cgi-bin/yabb2
chmod 755: cgi-bin/yabb2/AdminIndex.pl
chmod 755: cgi-bin/yabb2/FixFile.pl
chmod 666: cgi-bin/yabb2/Paths.pl
chmod 755: cgi-bin/yabb2/Setup.pl
chmod 755: cgi-bin/yabb2/YaBB.pl
chmod 777: cgi-bin/yabb2/Admin
chmod 666: cgi-bin/yabb2/Admin/* (all files)
chmod 777: cgi-bin/yabb2/Boards
chmod 666: cgi-bin/yabb2/Boards/* (all files)
chmod 777: cgi-bin/yabb2/Convert
chmod 777: cgi-bin/yabb2/Convert/Boards
chmod 777: cgi-bin/yabb2/Convert/Members
chmod 777: cgi-bin/yabb2/Convert/Messages
chmod 777: cgi-bin/yabb2/Convert/Variables
chmod 777: cgi-bin/yabb2/Help/English/Admin
chmod 777: cgi-bin/yabb2/Help/English/Admin/* (all files)
chmod 777: cgi-bin/yabb2/Help/English/Gmod
chmod 777: cgi-bin/yabb2/Help/English/Gmod/* (all files)
chmod 777: cgi-bin/yabb2/Help/English/Moderator
chmod 777: cgi-bin/yabb2/Help/English/Moderator/* (all files)
chmod 777: cgi-bin/yabb2/Help/English/User
chmod 777: cgi-bin/yabb2/Help/English/User/* (all files)
chmod 777: cgi-bin/yabb2/Languages/English
chmod 666: cgi-bin/yabb2/Languages/English/agreement.txt
chmod 666: cgi-bin/yabb2/Languages/English/censor.txt
chmod 777: cgi-bin/yabb2/Languages/English/* (all files)
chmod 777: cgi-bin/yabb2/Members
chmod 666: cgi-bin/yabb2/Members/* (all files)
chmod 777: cgi-bin/yabb2/Messages

```

```
chmod 666: cgi-bin/yabb2/Messages/* (all files)
chmod 777: cgi-bin/yabb2/Modules/Digest
chmod 777: cgi-bin/yabb2/Modules/Digest/HMAC_MD5.pm
chmod 777: cgi-bin/yabb2/Modules/Digest/MD5.pm
chmod 777: cgi-bin/yabb2/Modules/Time
chmod 777: cgi-bin/yabb2/Modules/Time/HiRes.pm
chmod 777: cgi-bin/yabb2/Modules/Upload
chmod 777: cgi-bin/yabb2/Modules/Upload/CGI.pm
chmod 777: cgi-bin/yabb2/Modules/Upload/CGI
chmod 777: cgi-bin/yabb2/Modules/Upload/CGI/Util.pm
chmod 766: cgi-bin/yabb2/Sources
chmod 755: cgi-bin/yabb2/Sources/* (all files)
chmod 766: cgi-bin/yabb2/Templates
chmod 766: cgi-bin/yabb2/Templates/default
chmod 666: cgi-bin/yabb2/Templates/default/* (all files)
chmod 766: cgi-bin/yabb2/Variables
chmod 666: cgi-bin/yabb2/Variables/* (all files)
chmod 777: public_html/yabbfiles
chmod 666: public_html/yabbfiles/*.js
chmod 777: public_html/yabbfiles/Attachments
chmod 777: public_html/yabbfiles/avatars
chmod 666: public_html/yabbfiles/avatars/* (all files)
chmod 777: public_html/yabbfiles/Buttons
chmod 666: public_html/yabbfiles/Buttons/English/* (all files)
chmod 777: public_html/yabbfiles/ModImages
chmod 777: public_html/yabbfiles/Smilies
chmod 666: public_html/yabbfiles/Smilies/* (all files)
chmod 777: public_html/yabbfiles/Templates/Admin
chmod 666: public_html/yabbfiles/Templates/Admin/default/* (all
files)
chmod 666: public_html/yabbfiles/Templates/Admin/default.css
chmod 777: public_html/yabbfiles/Templates/Forum
chmod 666: public_html/yabbfiles/Templates/Forum/default/* (all
files)
chmod 666: public_html/yabbfiles/Templates/Forum/default.css
```

- Uploading files and setting file permissions can be a long and tedious process. Once completed, however, the forum is almost ready to try. You can access the forum by navigating a web browser to `setup.pl` in the `cgi-bin`. It will be something like `http://www.example.com/cgi-bin/yabb2/setup.pl`.

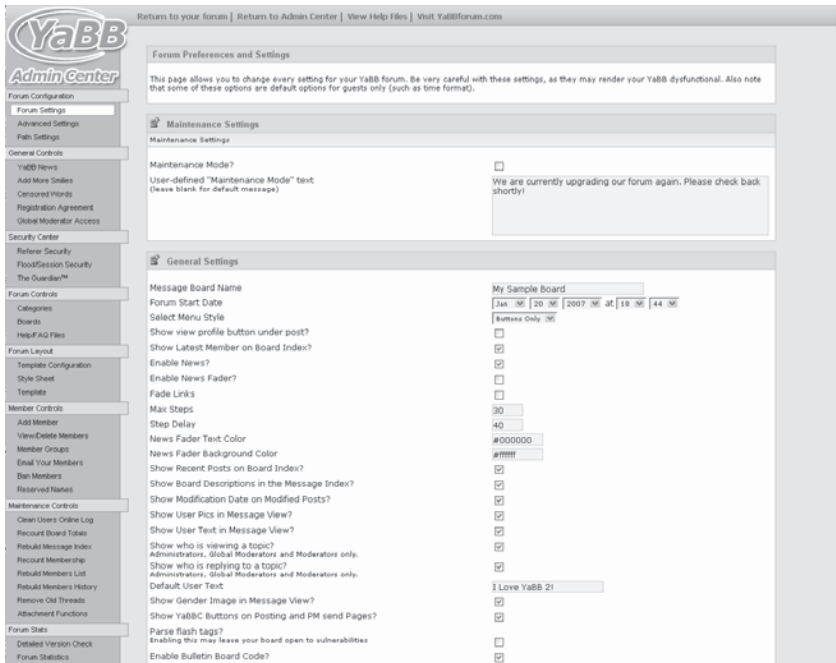


Figure 11-1

This page provides options to configure and accept forum settings. In this section, it's important to do the following:

- Enter a user name and password if asked. You begin as the administrator user. Your default user name will be `admin`, and the password will be `admin`. These settings should be changed as soon as possible.
- The paths screen displays where the YaBB HTML and data files are to be located. Default settings will be entered already, and often these are correct. In specific cases, these may need to be changed.

- c. Next, the localization screen allows administrators to set the time zone for the forum. The forum is now ready to use.

Hey, YaBB Administrator, you have 0 messages.

YaBB 2.1 Forum Software
Provided by YaBBForum.com

Feb 7th, 2007, 11:06am
News: Signup for free on our forum and benefit from new features!

Home Help Search Members Admin Profile Notification Logout

My Sample Board

Forum name	Last post	Topics	Posts
General Category			
<input checked="" type="checkbox"/> General Board This is the board for General Discussions. The board description can now hold multiple lines and can use HTML! Moderator: YaBB Administrator	→ Nov 6 th , 2005, 4:05pm Hi! Welcome to your new YaBB... By: YaBB Administrator	1	1
<input checked="" type="checkbox"/> Test Zone Test the forum out here. Posts made in this board will not add to your post count. Moderator: YaBB Administrator	→ N/A Hi By: N/A	0	0
Forum Staff			
<input type="checkbox"/> Global Announcements Topics you place in this board will display as a "global announcement" on the top of all other boards. Use this for things such as forum rules, top news articles, or important statements. Moderator: YaBB Administrator	→ N/A Hi By: N/A	0	0
<input type="checkbox"/> Recycle Bin If the Recycle Bin is turned on, removed topics will be moved to this board. This will allow you to recover them if it is necessary. You should purge messages in this board frequently to keep it clean.	→ N/A Hi By: N/A	0	0

New Posts No New Posts Mark All Posts As Read

Info Center

Forum Statistics

- Our users have made **1** Posts within **1** Topics.
The most recent post is: Welcome to your new YaBB 2.1 Forum! (Nov 6th, 2005, 4:05pm).
View the 10 most recent posts of this forum.
- We have **1** registered members.
The newest member is YaBB Administrator.
You have **0** Personal Messages.
- Most User** ever online was **2** on Feb 3rd, 2007, 10:07am.
- Most Members** ever online was **1** on Today at 11:06am.
- Most Guests** ever online was **1** on Today at 11:06am.

Users online

Members: **1**
YaBB Administrator

Guests: **0**

■ YaBB Administrator ■ Global Moderator

My Sample Board • Powered by YaBB 2.1
YaBB © 2000-2005. All Rights Reserved.

Figure 11-2

11.11 Distribution SDKs

11.11.1 NSIS

NSIS (Nullsoft Scriptable Install System) is an open-source library designed to create Windows installation packages for developers aiming to distribute their games to that audience.

Platforms supported: Windows

Web site: <http://nsis.sourceforge.net/>

License: Free for commercial and non-commercial use; see web site for more details.

11.11.2 Inno Setup

First created in 1997, Inno Setup is a freely available library to create Windows installation packages.

Platforms supported: Windows

Web site: <http://www.jrsoftware.org/isinfo.php>

License: Free for commercial and non-commercial use; see web site for more details.

11.11.2.1 Downloading, Installing, and Creating an Installer in Inno Setup

Inno Setup is a freely available library used by many game developers for creating an installation package (wizard) that installs their game to the local machine. It transfers the game files from a compressed archive on a CD or downloaded package to the local machine in a form that can execute successfully. The following step-by-step guide highlights how to download, install, and use Inno Setup to create installers for your own games.

1. Beginning from the Windows desktop, navigate a web browser to the Inno Setup home page at <http://www.jrsoftware.org/isinfo.php>.
2. At the Inno Setup home page, click either the **Downloads** link at the left of the page or the **Download Inno Setup** link on the page to display the Downloads page. There, download from the web to the local machine both the latest Inno Setup release and the Inno Setup Quick Start pack, and install each package to the local computer.
3. From the Windows Start menu, launch the newly installed Inno Setup IS Tool (Script Editor to create installation packages). Then from the IS Tool main menu, select **File | New** to create a new installation project (a project soon to be compiled into a completed installer, ready to run).

4. Enter the following script into the editor pane:

```
[Setup]
; This is a comment. The setup section describes the basic
properties
; of the installer; such as program title, version number,
default
; installation (destination) directory
AppName=My Program
AppVerName=My Program version 1.4
DefaultDirName={pf}\My Program
DefaultGroupName=My Program
OutputDir=C:\Ouput

[Files]
; Here list all files to be compiled into installation package;
files to be
; installed to the system by the installer at run-time
Source: c:\mystestpic.jpg; DestDir: {app}
```

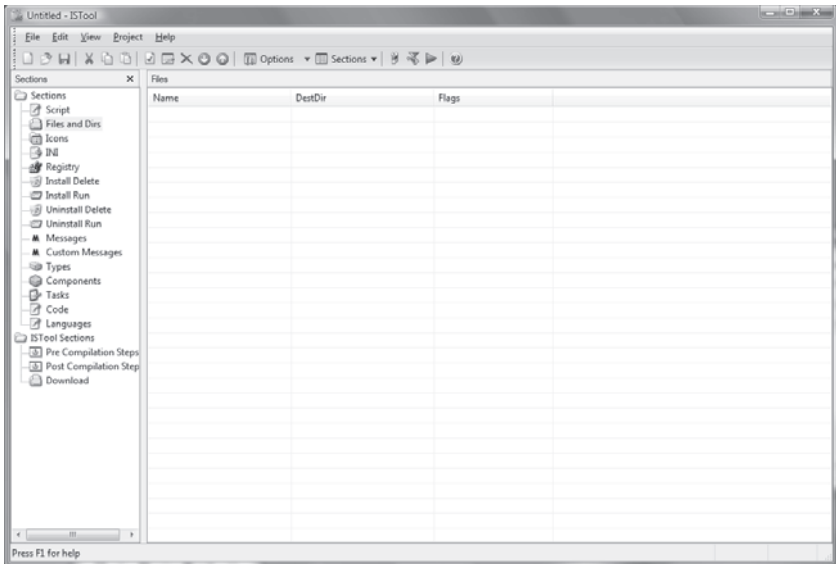


Figure 11-3

5. Then from the Inno Setup main menu, choose **Project | Compile Project** to generate a self-executing installation package. The installer is now compiled and ready to run.

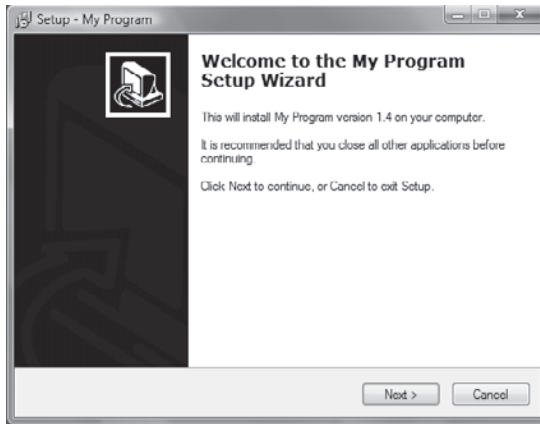


Figure 11-4

11.12 Conclusion

This chapter presented in summary a variety of SDKs (from ClanLib and ODE to NSIS and YaBB), many of them free and open-source, and each of them available to game developers looking to create cross-platform games.

This page intentionally left blank.

Appendix A

GNU Lesser General Public License

Version 3, 29 June 2007

Copyright (C) 2007 Free Software Foundation, Inc. <<http://fsf.org/>>

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

This version of the GNU Lesser General Public License incorporates the terms and conditions of version 3 of the GNU General Public License, supplemented by the additional permissions listed below.

0. Additional Definitions.

As used herein, “this License” refers to version 3 of the GNU Lesser General Public License, and the “GNU GPL” refers to version 3 of the GNU General Public License.

“The Library” refers to a covered work governed by this License, other than an Application or a Combined Work as defined below.

An “Application” is any work that makes use of an interface provided by the Library, but which is not otherwise based on the Library. Defining a subclass of a class defined by the Library is deemed a mode of using an interface provided by the Library.

A “Combined Work” is a work produced by combining or linking an Application with the Library. The particular version of the Library with which the Combined Work was made is also called the “Linked Version.”

The “Minimal Corresponding Source” for a Combined Work means the Corresponding Source for the Combined Work, excluding

any source code for portions of the Combined Work that, considered in isolation, are based on the Application, and not on the Linked Version.

The “Corresponding Application Code” for a Combined Work means the object code and/or source code for the Application, including any data and utility programs needed for reproducing the Combined Work from the Application, but excluding the System Libraries of the Combined Work.

1. Exception to Section 3 of the GNU GPL.

You may convey a covered work under sections 3 and 4 of this License without being bound by section 3 of the GNU GPL.

2. Conveying Modified Versions.

If you modify a copy of the Library, and, in your modifications, a facility refers to a function or data to be supplied by an Application that uses the facility (other than as an argument passed when the facility is invoked), then you may convey a copy of the modified version:

- a) under this License, provided that you make a good faith effort to ensure that, in the event an Application does not supply the function or data, the facility still operates, and performs whatever part of its purpose remains meaningful, or
- b) under the GNU GPL, with none of the additional permissions of this License applicable to that copy.

3. Object Code Incorporating Material from Library Header Files.

The object code form of an Application may incorporate material from a header file that is part of the Library. You may convey such object code under terms of your choice, provided that, if the incorporated material is not limited to numerical parameters, data structure layouts and accessors, or small macros, inline functions and templates (ten or fewer lines in length), you do both of the following:

- a) Give prominent notice with each copy of the object code that the Library is used in it and that the Library and its use are covered by this License.
-

- b) Accompany the object code with a copy of the GNU GPL and this license document.

4. Combined Works.

You may convey a Combined Work under terms of your choice that, taken together, effectively do not restrict modification of the portions of the Library contained in the Combined Work and reverse engineering for debugging such modifications, if you also do each of the following:

- a) Give prominent notice with each copy of the Combined Work that the Library is used in it and that the Library and its use are covered by this License.
- b) Accompany the Combined Work with a copy of the GNU GPL and this license document.
- c) For a Combined Work that displays copyright notices during execution, include the copyright notice for the Library among these notices, as well as a reference directing the user to the copies of the GNU GPL and this license document.
- d) Do one of the following:
 - 0) Convey the Minimal Corresponding Source under the terms of this License, and the Corresponding Application Code in a form suitable for, and under terms that permit, the user to recombine or relink the Application with a modified version of the Linked Version to produce a modified Combined Work, in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source.
 - 1) Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (a) uses at run time a copy of the Library already present on the user's computer system, and (b) will operate properly with a modified version of the Library that is interface-compatible with the Linked Version.
- e) Provide Installation Information, but only if you would otherwise be required to provide such information under section 6 of the GNU GPL, and only to the extent that such information is

necessary to install and execute a modified version of the Combined Work produced by recombining or relinking the Application with a modified version of the Linked Version. (If you use option 4d0, the Installation Information must accompany the Minimal Corresponding Source and Corresponding Application Code. If you use option 4d1, you must provide the Installation Information in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source.)

5. Combined Libraries.

You may place library facilities that are a work based on the Library side by side in a single library together with other library facilities that are not Applications and are not covered by this License, and convey such a combined library under terms of your choice, if you do both of the following:

- a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities, conveyed under the terms of this License.
- b) Give prominent notice with the combined library that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

6. Revised Versions of the GNU Lesser General Public License.

The Free Software Foundation may publish revised and/or new versions of the GNU Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library as you received it specifies that a certain numbered version of the GNU Lesser General Public License “or any later version” applies to it, you have the option of following the terms and conditions either of that published version or of any later version published by the Free Software Foundation. If the Library as you received it does not specify a version number of the GNU Lesser General Public License, you may choose any version of the GNU Lesser General Public License ever published by the Free Software Foundation.

If the Library as you received it specifies that a proxy can decide whether future versions of the GNU Lesser General Public License shall apply, that proxy's public statement of acceptance of any version is permanent authorization for you to choose that version for the Library.

This page intentionally left blank.

Appendix B

BSD License

Copyright (c) <year>, <copyright holder>

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met: Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution. Neither the name of the <organization> nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY <copyright holder> “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL <copyright holder> BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

This page intentionally left blank.

Appendix C

Creative Commons License

CREATIVE COMMONS CORPORATION IS NOT A LAW FIRM AND DOES NOT PROVIDE LEGAL SERVICES. DISTRIBUTION OF THIS LICENSE DOES NOT CREATE AN ATTORNEY-CLIENT RELATIONSHIP. CREATIVE COMMONS PROVIDES THIS INFORMATION ON AN “AS-IS” BASIS. CREATIVE COMMONS MAKES NO WARRANTIES REGARDING THE INFORMATION PROVIDED, AND DISCLAIMS LIABILITY FOR DAMAGES RESULTING FROM ITS USE.

License

THE WORK (AS DEFINED BELOW) IS PROVIDED UNDER THE TERMS OF THIS CREATIVE COMMONS PUBLIC LICENSE (“CCPL” OR “LICENSE”). THE WORK IS PROTECTED BY COPYRIGHT AND/OR OTHER APPLICABLE LAW. ANY USE OF THE WORK OTHER THAN AS AUTHORIZED UNDER THIS LICENSE OR COPYRIGHT LAW IS PROHIBITED.

BY EXERCISING ANY RIGHTS TO THE WORK PROVIDED HERE, YOU ACCEPT AND AGREE TO BE BOUND BY THE TERMS OF THIS LICENSE. THE LICENSOR GRANTS YOU THE RIGHTS CONTAINED HERE IN CONSIDERATION OF YOUR ACCEPTANCE OF SUCH TERMS AND CONDITIONS.

1. Definitions

1. “Collective Work” means a work, such as a periodical issue, anthology or encyclopedia, in which the Work in its entirety in unmodified form, along with a number of other contributions, constituting separate and independent works in themselves, are assembled into a collective whole. A work that constitutes a Collective Work will not be considered a Derivative Work (as defined below) for the purposes of this License.
2. “Derivative Work” means a work based upon the Work or upon the Work and other pre-existing works, such as a translation, musical arrangement, dramatization, fictionalization, motion picture version, sound recording, art reproduction, abridgment, condensation, or any other form in which the Work may be recast, transformed, or adapted, except that a work that constitutes a Collective Work will not be considered a Derivative Work for the purpose of this License.
3. “Licensor” means the individual or entity that offers the Work under the terms of this License.
4. “Original Author” means the individual or entity who created the Work.
5. “Work” means the copyrightable work of authorship offered under the terms of this License.
6. “You” means an individual or entity exercising rights under this License who has not previously violated the terms of this License with respect to the Work, or who has received express permission from the Licensor to exercise rights under this License despite a previous violation.

2. Fair Use Rights

Nothing in this license is intended to reduce, limit, or restrict any rights arising from fair use, first sale or other limitations on the exclusive rights of the copyright owner under copyright law or other applicable laws.

3. License Grant & Restrictions

Subject to the terms and conditions of this License, Licensor hereby grants You a worldwide, royalty-free, non-exclusive, perpetual (for the duration of the applicable copyright) license to exercise the rights in the Work as stated below on the conditions as stated below:

1. Re-creativity permitted. You may create and reproduce Derivative Works, provided that:
 - a. The Derivative Work(s) constitute a good-faith partial or recombined usage employing “sampling,” “collage,” “mash-up,” or other comparable artistic technique, whether now known or hereafter devised, that is highly transformative of the original, as appropriate to the medium, genre, and market niche; and
 - b. Your Derivative Work(s) must only make a partial use of the original Work, or if You choose to use the original Work as a whole, You must either use the Work as an insubstantial portion of Your Derivative Work(s) or transform it into something substantially different from the original Work. In the case of a musical Work and/or audio recording, the mere synchronization (“synching”) of the Work with a moving image shall not be considered a transformation of the Work into something substantially different.
2. You may distribute copies or phonorecords of, display publicly, perform publicly, and perform publicly by means of a digital audio transmission, any Derivative Work(s) authorized under this License.
3. Prohibition on advertising. All advertising and promotional uses are excluded from the above rights, except for advertisement and promotion of the Derivative Work(s) that You are creating from the Work and Yourself as the author thereof.
4. Noncommercial sharing of verbatim copies permitted.
 - a. You may reproduce the Work, incorporate the Work into one or more Collective Works, and reproduce the Work as incorporated in the Collective Works. You may distribute copies or

phonorecords of, display publicly, perform publicly, and perform publicly by means of a digital audio transmission the Work including or incorporated in Collective Works.

- b. You may not exercise any of the rights granted to You in the paragraph immediately above in any manner that is primarily intended for or directed toward commercial advantage or private monetary compensation. The exchange of the Work for other copyrighted works by means of digital file-sharing or otherwise shall not be considered to be intended for or directed toward commercial advantage or private monetary compensation, provided there is no payment of any monetary compensation in connection with the exchange of copyrighted works.

5. Attribution and Notice.

- a. If You distribute, publicly display, publicly perform, or publicly digitally perform the Work or any Derivative Works or Collective Works, You must keep intact all copyright notices for the Work and give the Original Author credit reasonable to the medium or means You are utilizing by conveying the name (or pseudonym if applicable) of the Original Author if supplied; the title of the Work if supplied; to the extent reasonably practicable, provide the Uniform Resource Identifier, if any, that Licensor specifies to be associated with the Work or a Derivative Work, unless such Uniform Resource Identifier does not refer to the copyright notice or licensing information for the Work; and in the case of a Derivative Work, provide a credit identifying the use of the Work in the Derivative Work (e.g., “Remix of the Work by Original Author,” or “Inclusion of a portion of the Work by Original Author in collage”). Such credit may be implemented in any reasonable manner; provided, however, that in the case of a Derivative Work or Collective Work, at a minimum such credit will appear where any other comparable authorship credit appears and in a manner at least as prominent as such other comparable authorship credit.
-

- b. You may distribute, publicly display, publicly perform or publicly digitally perform the Work only under the terms of this License, and You must include a copy of, or the Uniform Resource Identifier for, this License with every copy or phonorecord of the Work or Derivative Work You distribute, publicly display, publicly perform, or publicly digitally perform. You may not offer or impose any terms on the Work that alter or restrict the terms of this License or the recipients' exercise of the rights granted hereunder. You may not sublicense the Work. You must keep intact all notices that refer to this License and to the disclaimer of warranties. You may not distribute, publicly display, publicly perform, or publicly digitally perform the Work with any technological measures that control access of use of the Work in a manner inconsistent with the terms of this License. The above applies to the Work as incorporated in a Collective Work, but this does not require the Collective Work apart from the Work itself to be made subject to the terms of this License. Upon notice from any Licensor You must, to the extent practicable, remove from the Derivative Work or Collective Work any reference to such Licensor or the Original Author, as requested.

The above rights may be exercised in all media and formats whether now known or hereafter devised. The above rights include the right to make such modifications as are technically necessary to exercise the rights in other media and formats. All rights not expressly granted by Licensor are hereby reserved.

4. Disclaimer

UNLESS SPECIFIED OTHERWISE BY THE PARTIES IN A SEPARATE WRITING, LICENSOR OFFERS THE WORK AS-IS AND MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, WITHOUT LIMITATION, WARRANTIES OF TITLE, MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NONINFRINGEMENT, OR THE ABSENCE OF LATENT OR OTHER DEFECTS, ACCURACY, OR THE PRESENCE OR ABSENCE OF ERRORS, WHETHER OR NOT DISCOVERABLE.

5. Limitation on Liability

IN NO EVENT WILL LICENSOR BE LIABLE TO YOU ON ANY LEGAL THEORY FOR ANY SPECIAL, INCIDENTAL, CONSEQUENTIAL, PUNITIVE OR EXEMPLARY DAMAGES ARISING OUT OF THIS LICENSE OR THE USE OF THE WORK, EVEN IF LICENSOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

6. Termination

1. This License and the rights granted hereunder will terminate automatically upon any breach by You of the terms of this License. Individuals or entities who have received Derivative Works or Collective Works from You under this License, however, will not have their licenses terminated provided such individuals or entities remain in full compliance with those licenses. Sections 1, 2, 4, 5, 6, and 7 will survive any termination of this License.
2. Subject to the above terms and conditions, the license granted here is perpetual (for the duration of the applicable copyright in the Work). Notwithstanding the above, Licensor reserves the right to release the Work under different license terms or to stop distributing the Work at any time; provided, however that any such election will not serve to withdraw this License (or any other license that has been granted under the terms of this License), and this License will continue in full force and effect unless terminated as stated above.

7. Miscellaneous

1. Each time You distribute or publicly digitally perform the Work or a Collective Work, the Licensor offers to the recipient a license to the Work on the same terms and conditions as the license granted to You under this License.
 2. Each time You distribute or publicly digitally perform a Derivative Work, Licensor offers to the recipient a license to the original Work on the same terms and conditions as the license granted to You under this License.
-

3. If any provision of this License is invalid or unenforceable under applicable law, it shall not affect the validity or enforceability of the remainder of the terms of this License, and without further action by the parties to this agreement, such provision shall be reformed to the minimum extent necessary to make such provision valid and enforceable.
4. No term or provision of this License shall be deemed waived and no breach consented to unless such waiver or consent shall be in writing and signed by the party to be charged with such waiver or consent.
5. This License constitutes the entire agreement between the parties with respect to the Work licensed here. There are no understandings, agreements, or representations with respect to the Work, and with respect to the subject matter hereof, not specified above. Licensor shall not be bound by any additional provisions that may appear in any communication from You. This License may not be modified without the mutual written agreement of the Licensor and You.

This page intentionally left blank.

Appendix D

zlib/libpng License

Copyright (c) <year> <copyright holders>

This software is provided “as-is,” without any express or implied warranty. In no event will the authors be held liable for any damages arising from the use of this software.

Permission is granted to anyone to use this software for any purpose, including commercial applications, and to alter it and redistribute it freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not claim that you wrote the original software. If you use this software in a product, an acknowledgment in the product documentation would be appreciated but is not required.
2. Altered source versions must be plainly marked as such, and must not be misrepresented as being the original software.
3. This notice may not be removed or altered from any source distribution.

This page intentionally left blank.

Appendix E

The MIT License Template

Copyright (c) <year> <copyright holders>

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS,” WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

This page intentionally left blank.

Appendix F

STL Public Member Methods Reference

STD::STRING

`const_iterator begin () const`

`iterator begin ()`

`const _CharT * c_str () const`

`size_type capacity () const`

`void clear ()`

`int compare (size_type __pos, size_type __n1, const _CharT * __s, size_type __n2) const`

`int compare (size_type __pos, size_type __n1, const _CharT * __s) const`

`int compare (const _CharT * __s) const`

`int compare (size_type __pos1, size_type __n1, const basic_string & __str, size_type __pos2, size_type __n2) const`

`int compare (size_type __pos, size_type __n, const basic_string & __str) const`

`int compare (const basic_string & __str) const`

`size_type copy (_CharT * __s, size_type __n, size_type __pos=0) const`

`const _CharT * data () const`

`bool empty () const`

`const_iterator end () const`

iterator end ()

iterator erase (iterator __first, iterator __last)

iterator erase (iterator __position)

basic_string & erase (size_type __pos=0, size_type __n=npos)

size_type find (_CharT __c, size_type __pos=0) const

size_type find (const _CharT * __s, size_type __pos=0) const

size_type find (const basic_string & __str, size_type __pos=0) const

size_type find (const _CharT * __s, size_type __pos, size_type __n)
const

size_type find_first_not_of (_CharT __c, size_type __pos=0) const

size_type find_first_not_of (const _CharT * __s, size_type __pos=0)
const

size_type find_first_not_of (const _CharT * __s, size_type __pos,
size_type __n) const

size_type find_first_not_of (const basic_string & __str, size_type
__pos=0) const

size_type find_first_of (_CharT __c, size_type __pos=0) const

size_type find_first_of (const _CharT * __s, size_type __pos=0) const

size_type find_first_of (const _CharT * __s, size_type __pos, size_type
__n) const

size_type find_first_of (const basic_string & __str, size_type __pos=0)
const

size_type find_last_not_of (_CharT __c, size_type __pos=npow) const

size_type find_last_not_of (const _CharT * __s, size_type
__pos=npow) const

size_type find_last_not_of (const _CharT * __s, size_type __pos,
size_type __n) const

size_type find_last_not_of (const basic_string & __str, size_type
__pos=npow) const

size_type find_last_of (_CharT __c, size_type __pos=npow) const

size_type find_last_of (const _CharT * __s, size_type __pos=npow)
const

```
size_type find_last_of (const _CharT * __s, size_type __pos, size_type
    __n) const
size_type find_last_of (const basic_string & __str, size_type
    __pos=npos) const
allocator_type get_allocator () const
iterator insert (iterator __p, _CharT __c)
basic_string & insert (size_type __pos, size_type __n, _CharT __c)
basic_string & insert (size_type __pos, const _CharT * __s)
basic_string & insert (size_type __pos, const _CharT * __s, size_type
    __n)
basic_string & insert (size_type __pos1, const basic_string & __str,
    size_type __pos2, size_type __n)
basic_string & insert (size_type __pos1, const basic_string & __str)
template<class _InputIterator> void insert (iterator __p,
    _InputIterator __beg, _InputIterator __end)
void insert (iterator __p, size_type __n, _CharT __c)
size_type length () const
size_type max_size () const
basic_string & operator+= (_CharT __c)
basic_string & operator+= (const _CharT * __s)
basic_string & operator+= (const basic_string & __str)
basic_string & operator= (_CharT __c)
basic_string & operator= (const _CharT * __s)
basic_string & operator= (const basic_string & __str)
reference operator[] (size_type __pos)
const_reference operator[] (size_type __pos) const
void push_back (_CharT __c)
const_reverse_iterator rbegin () const
reverse_iterator rbegin ()
const_reverse_iterator rend () const
reverse_iterator rend ()
```

```
basic_string & replace (iterator __i1, iterator __i2, const_iterator
    __k1, const_iterator __k2)
basic_string & replace (iterator __i1, iterator __i2, iterator __k1,
    iterator __k2)
basic_string & replace (iterator __i1, iterator __i2, const_CharT
    *__k1, const_CharT *__k2)
basic_string & replace (iterator __i1, iterator __i2, _CharT *__k1,
    _CharT *__k2)
template<class _InputIterator> basic_string & replace (iterator __i1,
    iterator __i2, _InputIterator __k1, _InputIterator __k2)
basic_string & replace (iterator __i1, iterator __i2, size_type __n,
    _CharT __c)
basic_string & replace (iterator __i1, iterator __i2, const_CharT *__s)
basic_string & replace (iterator __i1, iterator __i2, const_CharT *__s,
    size_type __n)
basic_string & replace (iterator __i1, iterator __i2, const basic_string
    & __str)
basic_string & replace (size_type __pos, size_type __n1, size_type
    __n2, _CharT __c)
basic_string & replace (size_type __pos, size_type __n1, const_CharT
    *__s)
basic_string & replace (size_type __pos, size_type __n1, const_CharT
    *__s, size_type __n2)
basic_string & replace (size_type __pos1, size_type __n1, const
    basic_string & __str, size_type __pos2, size_type __n2)
basic_string & replace (size_type __pos, size_type __n, const
    basic_string & __str)
void reserve (size_type __res_arg=0)
void resize (size_type __n)
void resize (size_type __n, _CharT __c)
size_type rfind (_CharT __c, size_type __pos=npos) const
size_type rfind (const_CharT *__s, size_type __pos=npos) const
```

```

size_type rfind (const _CharT * __s, size_type __pos, size_type __n)
    const
size_type rfind (const basic_string & __str, size_type __pos=npos)
    const
size_type size () const
basic_string substr (size_type __pos=0, size_type __n=npos) const
void swap (basic_string & __s)

```

STD::VECTOR

```

void assign (size_type __n, const value_type & __val)
const_reference at (size_type __n) const
reference at (size_type __n)
const_reference back () const
reference back ()
const_iterator begin () const
iterator begin ()
size_type capacity () const
void clear ()
const_pointer data () const
pointer data ()
bool empty () const
const_iterator end () const
iterator end ()
iterator erase (iterator __first, iterator __last)
iterator erase (iterator __position)
const_reference front () const
reference front ()
template<typename _InputIterator> void insert (iterator __position,
    _InputIterator __first, _InputIterator __last)
void insert (iterator __position, size_type __n, const value_type & __x)
iterator insert (iterator __position, const value_type & __x)
size_type max_size () const

```

```
vector & operator= (const vector &__x)
const_reference operator[] (size_type __n) const
reference operator[] (size_type __n)
void pop_back ()
void push_back (const value_type &__x)
const_reverse_iterator rbegin () const
reverse_iterator rbegin ()
const_reverse_iterator rend () const
reverse_iterator rend ()
void reserve (size_type __n)
void resize (size_type __new_size, value_type __x=value_type())
size_type size () const
void swap (vector &__x)
template<typename _InputIterator> vector (_InputIterator __first,
    _InputIterator __last, const allocator_type &__a=allocator_type())
vector (const vector &__x)
vector (size_type __n, const value_type &__value=value_type(),
    const allocator_type &__a=allocator_type())
vector (const allocator_type &__a)
vector ()
~vector ()
```

Appendix G

SDL Key Codes

SDL Key	ASCII Value	Common Name
SDLK_BACKSPACE	\b	backspace
SDLK_TAB	\t	tab
SDLK_CLEAR		clear
SDLK_RETURN	\r	return
SDLK_PAUSE		pause
SDLK_ESCAPE	^[escape
SDLK_SPACE		space
SDLK_EXCLAIM	!	exclamation mark
SDLK_QUOTEDBL	"	double quote
SDLK_HASH	#	hash
SDLK_DOLLAR	\$	dollar
SDLK_AMPERSAND	&	ampersand
SDLK_QUOTE	'	quote
SDLK_LEFTPAREN	(left parenthesis
SDLK_RIGHTPAREN)	right parenthesis
SDLK_ASTERISK	*	asterisk
SDLK_PLUS	+	plus sign
SDLK_COMMA	,	comma
SDLK_MINUS	-	minus sign
SDLK_PERIOD	.	period
SDLK_SLASH	/	forward slash
SDLK_0	0	0
SDLK_1	1	1
SDLK_2	2	2

SDL Key	ASCII Value	Common Name
SDLK_3	3	3
SDLK_4	4	4
SDLK_5	5	5
SDLK_6	6	6
SDLK_7	7	7
SDLK_8	8	8
SDLK_9	9	9
SDLK_COLON	:	colon
SDLK_SEMICOLON	;	semicolon
SDLK_LESS	<	less-than sign
SDLK_EQUALS	=	equals sign
SDLK_GREATER	>	greater-than sign
SDLK_QUESTION	?	question mark
SDLK_AT	@	at
SDLK_LEFTBRACKET	[left bracket
SDLK_BACKSLASH	\	backslash
SDLK_RIGHTBRACKET]	right bracket
SDLK_CARET	^	caret
SDLK_UNDERSCORE	_	underscore
SDLK_BACKQUOTE	`	grave
SDLK_a	a	a
SDLK_b	b	b
SDLK_c	c	c
SDLK_d	d	d
SDLK_e	e	e
SDLK_f	f	f
SDLK_g	g	g
SDLK_h	h	h
SDLK_i	i	i
SDLK_j	j	j
SDLK_k	k	k
SDLK_l	l	l

SDL Key	ASCII Value	Common Name
SDLK_m	m	m
SDLK_n	n	n
SDLK_o	o	o
SDLK_p	p	p
SDLK_q	q	q
SDLK_r	r	r
SDLK_s	s	s
SDLK_t	t	t
SDLK_u	u	u
SDLK_v	v	v
SDLK_w	w	w
SDLK_x	x	x
SDLK_y	y	y
SDLK_z	z	z
SDLK_DELETE	^?	delete
SDLK_KP0		keypad 0
SDLK_KP1		keypad 1
SDLK_KP2		keypad 2
SDLK_KP3		keypad 3
SDLK_KP4		keypad 4
SDLK_KP5		keypad 5
SDLK_KP6		keypad 6
SDLK_KP7		keypad 7
SDLK_KP8		keypad 8
SDLK_KP9		keypad 9
SDLK_KP_PERIOD	.	keypad period
SDLK_KP_DIVIDE	/	keypad divide
SDLK_KP_MULTIPLY	*	keypad multiply
SDLK_KP_MINUS	-	keypad minus
SDLK_KP_PLUS	+	keypad plus
SDLK_KP_ENTER	\r	keypad enter
SDLK_KP_EQUALS	=	keypad equals

SDL Key	ASCII Value	Common Name
SDLK_UP		up arrow
SDLK_DOWN		down arrow
SDLK_RIGHT		right arrow
SDLK_LEFT		left arrow
SDLK_INSERT		insert
SDLK_HOME		home
SDLK_END		end
SDLK_PAGEUP		page up
SDLK_PAGEDOWN		page down
SDLK_F1		F1
SDLK_F2		F2
SDLK_F3		F3
SDLK_F4		F4
SDLK_F5		F5
SDLK_F6		F6
SDLK_F7		F7
SDLK_F8		F8
SDLK_F9		F9
SDLK_F10		F10
SDLK_F11		F11
SDLK_F12		F12
SDLK_F13		F13
SDLK_F14		F14
SDLK_F15		F15
SDLK_NUMLOCK		numlock
SDLK_CAPSLOCK		capslock
SDLK_SCROLLLOCK		scrolllock
SDLK_RSHIFT		right shift
SDLK_LSHIFT		left shift
SDLK_RCTRL		right ctrl
SDLK_LCTRL		left ctrl
SDLK_RALT		right alt

SDL Key	ASCII Value	Common Name
SDLK_LALT		left alt
SDLK_RMETA		right meta
SDLK_LMETA		left meta
SDLK_LSUPER		left windows key
SDLK_RSUPER		right windows key
SDLK_MODE		mode shift
SDLK_HELP		help
SDLK_PRINT		print screen
SDLK_SYSREQ		SysRq
SDLK_BREAK		break
SDLK_MENU		menu
SDLK_POWER		power
SDLK_EURO		euro

This page intentionally left blank.

Appendix H

Novashell Functions

A

Add

- BrainManager
- LayerList
- TextManager
- WatchManager
- AddApproach, GoalManager
- AddApproachAndSay, GoalManager
- AddBinding, InputManager
- AddCustom, TextManager
- AddCustomScreen, TextManager
- AddDelay, GoalManager
- AddForce, Entity
- AddForceAndTorque, Entity
- AddForceAndTorqueConstant, Entity
- AddForceConstant, Entity
- AddImageToMapCache, Entity
- Addition Operator
 - Rect
 - Rectf
 - Vector2
- AddMaterial, MaterialManager
- AddModPath, GameLogic
- AddMoveToMapAndPosition, GoalManager
- AddMoveToPosition, GoalManager
- AddNewGoal, GoalManager
- AddParticle, EffectBase
- AddRunScriptString, GoalManager

AddSay, GoalManager
AddSayByID, GoalManager
AskBrainByName, BrainManager
Assignment Operator, Vector2

B

BuildLocalNavGraph, Map

C

CalculateUnion
 Rect
 Rectf
CanWalkBetween, Entity
CanWalkTo, Entity
Clear, DataManager
ClearModPaths, GameLogic
Clone, Entity
Color, Color
Color(r,g,b,a), Color
ColorToString
CreateEffectExplosion, EffectManager
CreateEntity
 GlobalEntity
CreateEntitySpecial
CreateParticle, EffectManager
Cross, Vector2

D

Data
 Entity
 GameLogic
Delete, DataManager
Division Operator, Vector2
Dot, Vector2
DumpScriptInfo, Entity

E

Equality Operator, Vector2
Exists
 GlobalDataManager

F

FacingToVector
FunctionExists
GlobalEntity

G

Get
 DataManager
 LayerList
GetAcceleration, Entity
GetActiveMap, MapManager
GetActiveStateName, BrainManager
GetActiveZoneByMaterialType, Entity
GetAllLayers, LayerManager
GetAlpha, Color
GetAnimFrame, Entity
GetAnimPause, Entity
GetAsEntity, Tile
GetAttach, Entity
GetAttachOffset, Entity
GetAutoSave, Map
GetBaseColor
 Entity
 Tile
GetBlendMode, Entity
GetBlue, Color
GetBrainManager, Entity
GetCameraSettings, Camera
GetCollisionByRay, Map
GetCollisionLayers, LayerManager
GetCollisionRect, Entity
GetCollisionScale, Entity
GetCount
 LayerList
 TileList
GetCursorVisible, App
GetDistanceFromEntityByID, Entity
GetDistanceFromPosition, Entity

GetEnableRotationPhysics, Entity
GetEngineVersion, App
GetEngineVersionAsString, App
GetEntityByID
GetEntityByName
GetEntityByWorldPos
GetEntityIDByName
GetEntityTrackingByID, Camera
GetFacing, Entity
GetFacingTarget, Entity
GetFromString, TagManager
GetGameTick, App
GetGoalCount, GoalManager
GetGoalCountByName, GoalManager
GetGoalManager, Entity
GetGravityOverride, Entity
GetGreen, Color
GetHeight
 Rect
 Rectf
GetID
 Entity
 Tag
GetImageByID, Entity
GetImageClipRect, Entity
GetIsCreature, Entity
GetLayerID
 Entity
 Tile
GetLayerIDByName, LayerManager
GetLayerManager, Map
GetLinearVelocity, Entity
GetListenCollision, Entity
GetListenCollisionStatic, Entity
GetLockedScale, Entity
GetMap, Entity
GetMapName, Tag

GetMass, Entity
GetMaterial, MaterialManager
GetMousePos, InputManager
GetName
 Entity
 Map
GetNearbyZoneByMaterialType, Entity
GetNext, TileList
GetNum, DataManager
GetNumWithDefault, DataManager
GetOnGround, Entity
GetOnGroundAccurate, Entity
GetParticleByName, EffectManager
GetPersistent
 Entity
 Map
GetPlatform, App
GetPos
 Camera
 Entity
 Tag
 Tile
GetPosCentered, Camera
GetPosFromName, TagManager
GetRed, Color
GetRotation, Entity
GetRunUpdateEveryFrame, Entity
GetScale
 Camera
 Entity
GetSimulationSpeedMod, App
GetSizeX, Entity
GetSizeY, Entity
GetSpecial, Material
GetSpecialEntityByName
GetText, Entity
GetTextBounds, Entity

GetTextColor, Entity
GetTextScale, Entity
GetTick, App
GetTilesByRect, Map
GetTurnSpeed, Entity
GetType
 Material
 Tile
GetUserProfileName, GameLogic
GetVectorFacing, Entity
GetVectorFacingTarget, Entity
GetVectorToEntity, Entity
GetVectorToEntityID, Entity
GetVectorToPosition, Entity
GetVisibleLayers, LayerManager
GetWatchCount, WatchManager
GetWidth
 Rect
 Rectf
GetWithDefault, DataManager
GetWorldCollisionRect, Entity

H

HasLineOfSightToPosition, Entity

I

Inequality Operator, Vector2
InitCollisionDataBySize, Entity
InitGameGUI, GameLogic
InNearbyZoneByMaterialType, Entity
InstantUpdate, Camera
InState, BrainManager
InZoneByMaterialType, Entity
IsCloseToEntity, Entity
IsCloseToEntityByID, Entity
IsFacingTarget, Entity
IsGoalActiveByName, GoalManager
IsOnSameMapAsEntityByID, Entity

IsOverlapped
 Rect
 Rectf
IsPlaced, Entity
IsPlaying, SoundManager
IsShuttingDown, GameLogic
IsValidPosition, Entity

K

Kill, SoundManager

L

LastStateWas, BrainManager
Length, Vector2
Lerp
LoadCollisionInfo, Entity
LoadMapByName, MapManager
LogError
LogMsg

M

ModNum, DataManager
Multiply Operator, Vector2
MuteAll, SoundManager

N

Normalize, Vector2

P

ParmExists, App
Placeholder, SpecialEntity
Play, SoundManager
PlayLooping, SoundManager
PlaySound, Entity
PlaySoundPositioned, Entity

Q

Quit, GameLogic

R

Rect, Rect
Rect(left,top,right,bottom), Rect
Rect(Rect), Rect

Rectf, Rectf
Rectf(left,top,right,bottom), Rectf
Rectf(Rectf), Rectf
RectToString
RegisterAsWarp, TagManager
Remove
 BrainManager
 WatchManager
RemoveAllSubgoals, GoalManager
RemoveBinding, InputManager
RemoveBindingByEntity, InputManager
Reset, Camera
ResetNext, TileList
ResetUserProfile, GameLogic
RunFunction, Entity
RunFunctionIfExists, Entity
RunScript

S

Schedule
ScheduleSystem
ScreenToWorld
SendToBrainBase, BrainManager
SendToBrainByName, BrainManager
Set
 Color
 DataManager
Set1DAcceleration, MotionController
SetAcceleration, Entity
SetActiveMapByName, MapManager
SetAdditionalVector, EffectBase
SetAlpha, Color
SetAnimByName, Entity
SetAnimFrame, Entity
SetAnimPause, Entity
SetAttach, Entity
SetAttachOffset, Entity
SetAutoSave, Map

SetBaseColor
 Entity
 Tile

SetBlendMode, Entity

SetBlue, Color

SetCameraSettings, Camera

SetCollisionMode, Entity

SetCollisionScale, Entity

SetColor, Particle

SetCursorVisible, App

SetDampening, Entity

SetDefaultTalkColor, Entity

SetDeleteFlag, Entity

SetDensity, Entity

SetDesiredSpeed, Entity

SetEnableRotationPhysics, Entity

SetEntityTrackingByID, Camera

SetFacing, Entity

SetFacingTarget, Entity

SetGravityOverride, Entity

SetGreen, Color

SetHasPathNode, Entity

SetIfNull, DataManager

SetImage, Entity

SetImageByID, Entity

SetIsCreature, Entity

SetLayerID
 Entity
 Tile

SetLayerIDByName, Entity

SetListenCollision, Entity

SetListenCollisionStatic, Entity

SetLockedScale, Entity

SetMass, Entity

SetMaxMovementSpeed, Entity

SetMousePos, InputManager

SetMoveLerp, Camera

SetName, Entity
SetNavNodeType, Entity
SetNum, DataManager
SetNumIfNull, DataManager
SetOffset, EffectBase
SetOnGround, Entity
SetPan, SoundManager
SetPaused, SoundManager
SetPersistent
 Entity
 Map
SetPos
 Camera
 Entity
 Tile
SetPosAndMap, Entity
SetPosAndMapByTagName, Entity
SetPosCentered, Camera
SetPosCenteredTarget, Camera
SetPosTarget, Camera
SetPriority, SoundManager
SetRed, Color
SetRestartEngineFlag, GameLogic
SetRotation, Entity
SetRunUpdateEveryFrame, Entity
SetScale
 Camera
 Entity
SetScaleLerp, Camera
SetScaleTarget, Camera
SetScreenSize, App
SetSimulationSpeedMod, App
SetSpecial, Material
SetSpeedDistortion, EffectExplosion
SetSpeedFactor, SoundManager
SetSpriteByVisualStateAndFacing, Entity
SetStateByName, BrainManager

- SetText, Entity
- SetTextAlignment, Entity
- SetTextColor, Entity
- SetTextRect, Entity
- SetTextScale, Entity
- SetTrigger, Entity
- SetTurnSpeed, Entity
- SetType, Material
- SetUserProfileName, GameLogic
- SetVectorFacing, Entity
- SetVectorFacingTarget, Entity
- SetVisibilityNotifications, Entity
- SetVisualProfile, Entity
- SetVolume, SoundManager
- SetWindowTitle, App
- ShowMessage
- Stop, Entity
- StopX, Entity
- StopY, Entity
- StringToColor
- StringToRect
- StringToVector
- Subtraction Operator
 - Rect
 - Rectf
 - Vector2

T

- ToggleEditMode, GameLogic

U

- UnloadMapByName, MapManager
- UserProfileActive, GameLogic
- UserProfileExists, GameLogic

V

- VariableExists
 - Global
 - Entity
- Vector2, Vector2

Vector2(Vector2), Vector2

Vector2(x,y), Vector2

VectorToFacing

VectorToString

W

WorldToScreen

Appendix I

Director Events

- on activateApplication
- on activateWindow
- on beginSprite
- on closeWindow
- on cuePassed
- on deactivateApplication
- on deactivateWindow
- on DVDeventNotification
- on endSprite
- on enterFrame
- on EvalScript
- on exitFrame
- on getBehaviorDescription
- on getBehaviorTooltip
- on getPropertyDescriptionList
- on hyperlinkClicked
- on idle
- on isOKToAttach
- on keyDown
- on keyUp
- on mouseDown
- on mouseEnter
- on mouseLeave

on mouseUp
on mouseUpOutside
on mouseWithin
on moveWindow
on openWindow
on prepareFrame
on prepareMovie
on resizeWindow
on rightMouseDown
on rightMouseUp
on runPropertyDialog
on savedLocal
on sendXML
on startMovie
on stepFrame
on stopMovie
on streamStatus
on timeOut
on zoomWindow
trayIconMouseDoubleClick
trayIconMouseDown
trayIconRightMouseDown

Appendix J

OGRE OIS Key Codes

KC_UNASSIGNED	= 0x00	
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
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/Germany // 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 Portugese (Brazilian) // keyboards
KC_CONVERT	= 0x79	// (Japanese keyboard)
KC_NOCONVERT	= 0x7B	// (Japanese keyboard)
KC_YEN	= 0x7D	// (Japanese keyboard)
KC_ABNT_C2	= 0x7E	// Numpad . on Portugese // (Brazilian) keyboards
KC_NUMPADEQUALS	= 0x8D	// = on numeric keypad // (NEC PC98)
KC_PREVTRACK	= 0x90	
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

Index

2D games, 225
3D games, 225, 299-302

A

alpha channel, 102
alpha transparency, 237
AngelCode, 350
arrays, 123-124
artificial intelligence SDKs, 336-337,
347-348
Audacity, 83-84, 176-177
 downloading and installing on Mac,
 180-181
 downloading and installing on Ubuntu,
 178-179
 downloading and installing on Win-
 dows, 180-181
Audiere, 343
audio, 173
 programming with SDL_mixer, 186-187
 SDKs, 336, 342-343
 tiles, 250-251
Automatix, 70
 installing and using on Ubuntu, 71-73

B

background music, 173-174
base class, 202
BASH shell commands, 74-78
 creating and compiling C program with,
 79-80
BASS, 342
Blender 3D, 83, 109
 creating mesh in, 325
 installing on Mac, 111-113
 installing on Ubuntu, 109-111
 installing on Windows, 111-113
blitting, 160

Boost, 347
BSD License, 369
budget, 117
Bullet, 345

C

cast members, 281-282
 animating sprites using, 292-293
casual games, 121-122
Cedega, 66
channels, 289
 handling, 198-200
char*, converting strings to, 129-130
character, moving using keyboard input,
258-261
ClanLib, 339
 installing, 339-342
classes, scripting, 289-290
Code::Blocks, 81-82, 84-85
 configuring project to use STL with,
 126
 downloading and configuring SDL
 Image Development Libraries in,
 163-167
 downloading and installing in
 Ubuntu, 86-89
 downloading and installing in
 Windows, 90-95
 Hello World application, 98-100
 using, 95-97
 using to create SDL project in
 Ubuntu, 143-145
 using to create SDL project in
 Windows, 148-152
collision detection, 248
color keying, 168
color tiles, 251

coordinates, 213

Creative Commons License, 371-377

CrossOver, 67

cross-platform, 1-2

 game development, 15-17

 games, 2-3, 11-14

 SDKs, 335-338

 software, 2

 tools, 81-84

D

data types, 122-123

depth sorting, 222

Director, 267, 268, 295

 building stand-alone game in, 297

 building web game in, 296

 cast members, 281-282

 classes for scripting, 289-290

 components of, 281-285

 cross-compatibility of, 271-272

 downloading and installing, 272-274

 events, 407-408

 games, 269-271

 Hello World application, 274-280

 scripting with JavaScript, 285-288

 Stage, 282-283

 Time-Line window, 284-285

distribution SDKs, 338, 358-361

distributions, 2

double buffering, 159

E

entity, 237

 attaching script to, 254-256

 intelligent, 255

 layer, 240

 resources, 238

event, 157

 polling, 157-158

F

FANN, 348

Firefox, 60-61

first-person shooter, *see* FPS

Flash, 267

Flash Player, 267

FMOD, 342

FPS, 119

frame events, receiving, 318-321

frame scripts, 286-287

FreeSound projects, 175-176

Freespire, 10

G

Game Editor, 352

game engines, 337, 351-352

game loop, 124, 134-136, 154

game object,

 creating, 202-203

 creating derived, 204

 maintaining, 205-206

game programming, 116

game world, 202

GameMonkey, 351

games, 201

 2D, 225

 3D, 225, 299-302

 components of, 117

 considerations when creating, 122-124

 creating with Novashell, 243-245

 cross-platform, 2-3, 11-14

 installing and playing on Ubuntu, 63-65

 OGRE 3D, 304-306

Garfixia AI Repository, 348

genres, 117, 118-122

GIMP, 62, 82, 101-102

 installing, 102-103

 using to create tileable textures,

 103-105

 using to edit image transparency,

 106-108

global event scripts, 287-288

GNU Lesser General Public License,

 363-367

graphics SDKs, 335-336, 338-342

guest, 41

GUI SDKs, 337, 352

H

hard disks, formatting, 18

hardware acceleration, 303

-
- HawkNL, 346
 - Hello World application,
 - in Code::Blocks, 98-100
 - in Director, 274-280
 - hidden layers, 240
 - hierarchy, relationship, 219-220
 - host, 41
 - I**
 - identifier, 202
 - image files, importing into Novashell, 246
 - image transparency, editing with GIMP, 106-108
 - incidental music, 173-174
 - Inno Setup, 359
 - creating installer with, 359-361
 - input SDKs, 337, 348-349
 - installer, creating with Inno Setup, 359-361
 - Invisible Wall tiles, 251
 - irrKlang, 343
 - Irrlicht, 351
 - ISO, burning Ubuntu, 8
 - J**
 - JavaScript, using with Director, 285-288
 - K**
 - key codes,
 - OGRE OIS, 409-414
 - SDL, 389-393
 - keyboard input, using to move character, 258-261
 - L**
 - levels, editing with Novashell, 241-243
 - LibGIL, 348
 - licenses,
 - BSD License, 369
 - Creative Commons License, 371-377
 - GNU Lesser General Public License, 363-367
 - MIT License template, 381
 - zlib/libpng License, 379
 - lights, adding, 325-328
 - Linux, 5-6
 - distributions, 7-10
 - downloading Novashell for, 228-230
 - shell, 73
 - Ubuntu, *see* Ubuntu
 - list, 124
 - creating, 131
 - printing, 291-292
 - working with, 132-134
 - local event scripts, 288
 - LookAt vector, 216-217
 - Lua, 349
 - M**
 - Mac,
 - downloading and installing Audacity on, 180-181
 - downloading and installing Schism Tracker on, 185
 - downloading Novashell for, 228-230
 - installing Blender 3D on, 111-113
 - installing GIMP on, 102-103
 - Mac OS X, 4-5
 - magnitude, 214
 - maps, 202, 207-208
 - implementing, 208-209
 - editing with Novashell, 241-243
 - mesh, 322
 - creating in Blender 3D, 325
 - message pump, 124
 - Microsoft Windows, *see* Windows
 - Microsoft Windows Vista, *see* Windows Vista
 - Microsoft Windows XP, *see* Windows XP
 - MIT License template, 381
 - mouse events, querying, 294
 - multiple booting setup, 16, 17, 19-40
 - multiple machine setup, 15
 - music, 173-174, 192
 - controlling playback of, 195-197
 - loading, 193
 - playing, 194-195
 - software, 183
 - sources of, 182
-

N

- network SDKs, 336, 346-347
- Newton Game Dynamics, 344
- node, 322
- Novashell, 225-227
 - creating path node network with, 262-264
 - creating player entity with, 247
 - detecting collisions with, 248-249
 - downloading, 228-230
 - functions, 395-406
 - games, 230-233
 - importing image files into, 246
 - modes, 234
 - scripting in, 252-253
 - System Palette, 249-250
 - tile types, 249-250
 - using, 233-236
 - using to create game, 243-245
- Novashell Console, 253
- Novashell editor, 236-237
 - tools, 239-240
 - using, 241-243
- NSIS, 358

O

- objective, 117
- objects, adding to scene, 321-324
- ODE, 97, 344
- OGRE 3D, 302-304
 - downloading and installing on Ubuntu, 307-311
 - downloading and installing on Windows, 312-315
 - games, 304-306
 - installing, 306
 - reading user input with, 328-333
 - sample application, 315-317
- OGRE OIS key codes, 409-414
- online forum, creating, 353-358
- OPAL, 345
- OpenAL, 343
- OpenGL, 97, 338
- OpenGUI, 352
- OpenInput, 349

- OpenOffice.org, 61
- OpenSteer, 347

P

- page flipping, 158-159
- painter's algorithm, 222
- particle systems, adding, 325-328
- path node network, creating using Novashell, 262-264
- Path nodes, 252
- pathfinding, 252, 262
- physics SDKs, 336, 344-345
- PhysX, 345
- pixels, drawing, 290-291
- platformer, 119-120
- platforms, 2
 - Linux, 5-6
 - Mac OS X, 4-5
 - Microsoft Windows, 3-4
- player entity, creating with Novashell, 247
- PNG file, loading SDL surface from, 166-167, 169
- prerendered 3D games, 301-302
- PTK, 338-339
- Python, 349-350

R

- RakNet, 346
- real-time 3D games, 299-301
- real-time strategy, *see* RTS
- relationship hierarchy, 219-220
 - supporting, 220-221
- resources, 237
- role-playing game, *see* RPG
- root node, 322
- RPG, 118
- RTS, 120-121
- Ruby, 350

S

- samples, 192
 - loading, 198
 - playing, 197-198
- scene, 322
 - adding objects to, 321-324

-
- Schism Tracker,
 - downloading and installing on Mac, 185
 - downloading and installing on Ubuntu, 183-184
 - downloading and installing on Windows, 185
 - Script tiles, 252
 - scripting,
 - classes for, 289-290
 - in Novashell, 252-253
 - SDKs, 337, 349-351
 - scripts,
 - attaching to entity, 254-256
 - types of, 286-288
 - SDKs,
 - artificial intelligence, 336-337, 347-348
 - audio, 336, 342-343
 - cross-platform, 335-338
 - distribution, 338, 358-361
 - graphics, 335-336, 338-342
 - GUI, 337, 352
 - input, 337, 348-349
 - network, 336, 346-347
 - physics, 336, 344-345
 - scripting, 337, 349-351
 - web, 337, 352-358
 - SDL, 97, 137
 - downloading and installing on Ubuntu, 140-141
 - downloading and installing on Windows, 146-148
 - downloading documentation for, 142
 - example application, 169-172
 - initializing, 153
 - key codes, 389-393
 - subsystems, 138-139
 - uninitializing, 153
 - SDL Image Development Libraries,
 - downloading and configuring on Ubuntu, 163-165
 - downloading and configuring on Windows, 165-167
 - SDL project,
 - creating in Ubuntu using Code::Blocks, 143-145
 - creating in Windows using Code::Blocks, 148-152
 - SDL_mixer,
 - handling audio with, 192-200
 - initializing, 190-192
 - installing and configuring on Ubuntu, 187-189
 - installing and configuring on Windows, 189-190
 - programming audio with, 186-187
 - SDL_net, 347
 - Shockwave, 14, 267
 - cross-compatibility of, 271-272
 - Simple DirectMedia Layer, *see* SDL
 - SLAX, 9
 - software,
 - cross-platform, 2
 - free, 6
 - music, 183
 - open-source, 6
 - sound effects, 176-177
 - sound, 173
 - loading as samples, 198
 - sound effects, 175, 177
 - software, 176-177
 - sprites, 289
 - animating, 292-293
 - printing list of, 291-292
 - Squirrel, 350
 - Stage, 282-283
 - stand-alone games, building in Director, 297
 - Standard Template Library, *see* STL
 - std::string, 125
 - public member methods, 383-387
 - working with, 126-130
 - std::vector, 130-131
 - public member methods, 387-388
 - working with, 131-134
 - STL, 125
 - configuring projects to use, 126
 - public member methods, 383-388
 - strings, 125
 - working with, 126-130
-

surfaces, 157, 159-160
 blitting, 160-161
 color keying, 168-169
 loading, 166-167
 loading from PNG file, 166-167, 169
 optimizing, 161-162

T

TBS, 120-121
 texture tile set, 237
 textures, 237
 creating tileable with GIMP, 103-105
 seamless, 101
 stochastic, 101
 tileable, 103
 tile resources, 237-238
 tile set, 208
 loading, 209-211
 tiles, 208, 237
 animating, 211
 moveable, 212
 static, 212
 subdividing, 247-248
 using vector to move, 215-218
 Time-Line window, 284-285
 tools, cross-platform, 81-84
 Torque, 351
 transformations, 218
 transgaming, 65-66
 True Axis, 344
 turn-based strategy, *see* TBS

U

Ubuntu, 7
 Add/Remove Applications panel, 57
 automating, 70
 components, 52-62
 creating SDL project using
 Code::Blocks in, 143-145
 creating virtual machine for, 43-47
 desktop, 54
 downloading and burning as ISO, 8
 downloading and configuring SDL
 Image Development Libraries on,
 163-165

 downloading and installing Audacity on,
 178-179
 downloading and installing
 Code::Blocks on, 86-89
 downloading and installing OGRE 3D
 on, 307-311
 downloading and installing Schism
 Tracker on, 183-184
 downloading and installing SDL on,
 140-141
 downloading SDL documentation for,
 142
 installing, 33-39
 installing and configuring SDL_mixer
 on, 187-189
 installing and playing a game on, 63-65
 installing and using Automatix on,
 71-73
 installing Blender 3D on, 109-111
 installing Wine on, 68-70
 Restricted Drivers Manager, 56
 Synaptic Package Manager, 58
 System Monitor, 55
 troubleshooting, 50-52
 Update Manager, 55-56
 using VBWare Workstation with, 43-47
 Ubuntu Terminal, 59
 creating and compiling C program with,
 79-80
 unit vector, 214
 user input, reading with OGRE, 328-333

V

vector, 213
 normalized, 214
 unit, 214
 using to move tiles, 215-218
 virtual machine, 266
 creating for Ubuntu, 43-47
 software, 267
 Virtual PC, 41
 virtualization, 16-17, 40-41
 Vista, *see* Windows Vista
 visual profile, 256-258

VMWare Workstation, 41-42
using with Ubuntu, 43-47

W

Warp nodes, 252
waypoint, 262-263
Waypoint nodes, 252
web games, 265-266
building in Director, 296
web SDKs, 337, 352-358
windows, 154
creating, 155-156
Windows, 3-4
creating SDL project using
Code::Blocks in, 148-152
downloading and configuring SDL
Image Development Libraries on,
165-167
downloading and installing Audacity on,
180-181
downloading and installing
Code::Blocks in, 90-95
downloading and installing Director on,
272-274
downloading and installing OGRE 3D
on, 312-315

downloading and installing Schism
Tracker on, 185

downloading and installing SDL on,
146-148

downloading Novashell for, 228-230

installing and configuring SDL_mixer
on, 189-190

installing Blender 3D on, 111-113

installing GIMP on, 102-13

Windows Vista,

installing, 27-33

troubleshooting in Ubuntu, 51-52

Windows XP, installing, 20-26

Wine, 68

installing on Ubuntu, 68-70

X

XML entity profile, 256-258

Y

YaBB, 352

creating online forum with, 353-358

Z

zlib/libpng License, 379

z-order, 222-223

z-ordering, 240
